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

IpParTTL/IpPar_1/ IpPar_2/IpPar_3/ IpPar_4/IpPar_5/ IpPar485

Driver Documentation

Win32 Driver Model

Revision B
Corresponding Hardware: Revision B
10-2001-0102

lpParTTL/_1/_2/_3/_4/_5/485
WDM Device Drivers for the
IP-Parallel-IO IP Modules

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

Manual Revision C. Revised October 27, 2009.

Table of Contents

Introduction 5

Note 5

Driver Installation 5

Windows 2000 Installation 6

Windows XP Installation 6

Driver Startup 7

IO Controls 10
- IOCTL_IPPAR??_GET_INFO 10
- IOCTL_IPPAR??_SET_IP_CONTROL 10
- IOCTL_IPPAR??_GET_IP_CONTROL 10
- IOCTL_IPPAR??_SET_BASE_CONFIG 11
- IOCTL_IPPAR??_GET_BASE_CONFIG 11
- IOCTL_IPPAR??_SET_TTL_DATA 11
- IOCTL_IPPAR??_GET_TTL_DATA 11
- IOCTL_IPPAR??_SET_485_DIR 11
- IOCTL_IPPAR??_GET_485_DIR 12
- IOCTL_IPPAR??_SET_485_DATA 12
- IOCTL_IPPAR??_GET_485_DATA 12
- IOCTL_IPPAR??_SET_TTL_INT_EN 12
- IOCTL_IPPAR??_GET_TTL_INT_EN 12
- IOCTL_IPPAR??_SET_485_INT_EN 13
- IOCTL_IPPAR??_GET_485_INT_EN 13
- IOCTL_IPPAR??_SET_TTL_EDGE_LEVEL 13
- IOCTL_IPPAR??_GET_TTL_EDGE_LEVEL 13
- IOCTL_IPPAR??_SET_485_EDGE_LEVEL 13
- IOCTL_IPPAR??_GET_485_EDGE_LEVEL 14
- IOCTL_IPPAR??_SET_TTL_POLARITY 14
- IOCTL_IPPAR??_GET_TTL_POLARITY 14
- IOCTL_IPPAR??_SET_485_POLARITY 14
- IOCTL_IPPAR??_GET_485_POLARITY 14
- IOCTL_IPPAR??_READ_DIRECT 15
- IOCTL_IPPAR??_READ_ FILTERED 15
- IOCTL_IPPAR??_SET_COUNTER_PRELOAD 15
- IOCTL_IPPAR??_GET_COUNTER_PRELOAD 15
- IOCTL_IPPAR??_SET_TIMER_MASK 15
- IOCTL_IPPAR??_GET_TIMER_MASK 16
- IOCTL_IPPAR??_LOAD_COUNTER 16
- IOCTL_IPPAR??_CLEAR_TIMER 16
- IOCTL_IPPAR??_GET_TIMER_COUNT 16
- IOCTL_IPPAR??_GET_STATUS 16

IOCTL_IPPAR??_REGISTER_EVENT 17
IOCTL_IPPAR??_ENABLE_INTERRUPT 17
IOCTL_IPPAR??_DISABLE_INTERRUPT 17
IOCTL_IPPAR??_FORCE_INTERRUPT 17
IOCTL_IPPAR??_SET_VECTOR 18
IOCTL_IPPAR??_GET_VECTOR 18
IOCTL_IPPAR??_GET_ISR_STATUS 18

WARRANTY AND REPAIR 19

Service Policy 19
Out of Warranty Repairs 19

For Service Contact: 19

Introduction

Note: In this document IpPar?? and IP- Parallel-?? refer to one of seven versions of the IpParIO driver and IP-Parallel-IO hardware. The versions and the I/O distributions are as follows:

Board Type	Driver Name	IO configuration
IP-Parallel-TTL	IpParTTL	48 TTL / 0 RS485
IP-Parallel-1	IpPar_1	40 TTL / 4 RS485
IP-Parallel-2	IpPar_2	32 TTL / 8 RS485
IP-Parallel-3	IpPar_3	24 TTL / 12 RS485
IP-Parallel-4	IpPar_4	16 TTL / 16 RS485
IP-Parallel-5	IpPar_5	8 TTL / 20 RS485
IP-Parallel-485	IpPar485	0 TTL / 24 RS485

The IpPar?? driver is a Win32 driver model (WDM) device driver for the IP-Parallel-?? board from Dynamic Engineering. Each IP- Parallel-?? board implements a parallel interface using I/O driver standards of either TTL, RS485 or a combination of both. A separate Device Object controls each IP- Parallel-?? board, and a separate handle references each Device Object. IO Control calls (IOCTLs) are used to configure the board and to transfer data to and from the IP device's parallel interface.

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

Warning: 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 IpPar??.sys, IpDevice.inf, DDIpPar??.h, IpPar??GUID.h, IPP??Test.exe, and IPP??Test source files.

DDIpPar??.h is the C header file that defines the Application Program Interface (API) to the driver. IpPar??GUID.h is a C header file that defines the device interface identifier for the IpPar??. These files are required at compile time by any application that wishes to interface with an IpPar?? driver, but are not needed for driver installation.

IPP??Test.exe is a sample Win32 console application that makes calls into the IpPar?? 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 ***IPP??Test –d0 -?*** to display a list of commands (the IPP??Test.exe file must be in the

directory that the window is referencing). The commands are all of the form ***IPP??Test –dn –im*** where ***n*** and ***m*** are the device number and driver ioctl number respectively. This application is intended to test the proper functioning of the driver calls, not for normal hardware operation.

Windows 2000 Installation

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

With the 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 IpPar??.sys to a floppy disk, or CD if preferred.

With the 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.

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 IpPar??GUID.h.

Below is example code for opening a handle for device 0. The device number is underlined and italicized in the SetupDiEnumDeviceInterfaces call.

```
// The maximum length of the device name for a given interface
#define MAX_DEVICE_NAME 256
// Device number
ULONG                              devNum
// Device interface index
ULONG                              i;
// Handle to the device object
HANDLE                             hIpPar?? =  INVALID_HANDLE_VALUE;
// Return status from command
LONG                               status;
// Handle to device interface information structure
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 structures for this device
SP_DEVICE_INTERFACE_DATA           interfaceData;
PSP_DEVICE_INTERFACE_DETAIL_DATA pDeviceDetail;

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

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

// 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 and Create the handle to the device
hIpPar?? = CreateFile(deviceName,
                      GENERIC_READ     | GENERIC_WRITE,
                      FILE_SHARE_READ  | FILE_SHARE_WRITE,
                      NULL,
                      OPEN_EXISTING,
                      NULL,
                      NULL);

if(hIpPar?? == 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 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 for the IpPar?? driver are described below:

IOCTL_IPPAR??_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 driver parameters. See DDIpPar??.h for the definition of DEVICE_IP_DEVICE_INFO.

IOCTL_IPPAR??_SET_IP_CONTROL

Function: Sets the configuration of the IP slot.
Input: unsigned long int
Output: None
Notes: Controls the IP clock speed and interrupt enables for the IP slot that the board occupies. See the bit definitions in the DDIpPar??.h header file for more information.

IOCTL_IPPAR??_GET_IP_CONTROL

Function: Returns the configuration of the IP slot.
Input: None
Output: unsigned long int
Notes: Returns the slot configuration register value for the IP slot that the board occupies. See the bit definitions in the DDIpPar??.h header file for more information.

IOCTL_IPPAR??_SET_BASE_CONFIG

Function: Sets configuration parameters in the base control register.
Input: IPPARIO_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.

IOCTL_IPPAR??_GET_BASE_CONFIG

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

IOCTL_IPPAR??_SET_TTL_DATA

Function: Sets the value of the TTL outputs on the board.
Input: IPPARIO_TTL_BITS structure
Output: None
Notes: The number of valid bits varies with the number of TTL outputs on the board. This call is not valid on the IP-Parallel-485 board as it has no TTL drivers.

IOCTL_IPPAR??_GET_TTL_DATA

Function: Returns the state of the TTL outputs in the output data register.
Input: None
Output: IPPARIO_TTL_BITS structure
Notes: The number of valid bits varies with the number of TTL outputs. This call is not valid on the IP-Parallel-485 board as it has no TTL drivers.

IOCTL_IPPAR??_SET_485_DIR

Function: Sets the directions for the RS485 drivers on the board.
Input: unsigned long int
Output: None
Notes: The number of valid bits varies with the number of RS485 drivers from 24 on the IP-Parallel-485 to zero on the IP-Parallel-TTL. A one in a bit position corresponds to setting that driver to be an output, while a zero corresponds to an input. This call is not valid on the IP-Parallel-TTL board.

IOCTL_IPPAR??_GET_485_DIR

Function: Returns the direction bits for the RS485 drivers on the board.
Input: None
Output: unsigned long int
Notes: Returns the bits set in the previous call. This call is not valid on the IP-Parallel-TTL board.

IOCTL_IPPAR??_SET_485_DATA

Function: Sets the value of the RS485 outputs on the board.
Input: unsigned long int
Output: None
Notes: The number of valid bits varies with the number of RS485 drivers on the board and only the drivers that are set to be outputs can drive the output bus. This call is not valid on the IP-Parallel-TTL board as it has no RS485 drivers.

IOCTL_IPPAR??_GET_485_DATA

Function: Returns the state of the RS485 bits in the output data register.
Input: None
Output: unsigned long int
Notes: The number of valid bits varies with the number of RS485 drivers. This call is not valid on the IP-Parallel-TTL board as it has no RS485 drivers.

IOCTL_IPPAR??_SET_TTL_INT_EN

Function: Selects which TTL inputs can cause an interrupt.
Input: IPPARIO_TTL_BITS structure
Output: None
Notes: This call defines the mask of which of the TTL input lines will be enabled to cause an interrupt when the specified conditions are met (1 = enabled, 0 = disabled). The number of valid bits varies with the number of TTL I/O. This call is not valid on the IP-Parallel-485 board as it has no TTL drivers.

IOCTL_IPPAR??_GET_TTL_INT_EN

Function: Returns the interrupt enable values set in the previous call.
Input: None
Output: IPPARIO_TTL_BITS structure
Notes: The number of valid bits varies with the number of TTL I/O lines. This call is not valid on the IP-Parallel-485 board as it has no TTL drivers.

IOCTL_IPPAR??_SET_485_INT_EN

Function: Selects which RS485 inputs can cause an interrupt.
Input: unsigned long int
Output: None
Notes: This call defines the mask of which of the RS485 input lines will be enabled to cause an interrupt when the specified conditions are met (1 = enabled, 0 = disabled). The number of valid bits varies with the number of RS485 drivers. This call is not valid on the IP-Parallel-TTL board as it has no RS485 drivers.

IOCTL_IPPAR??_GET_485_INT_EN

Function: Returns the interrupt enable value set in the previous call.
Input: None
Output: unsigned long int
Notes: The number of valid bits varies with the number of RS485 I/O lines. This call is not valid on the IP-Parallel-TTL board as it has no RS485 drivers.

IOCTL_IPPAR??_SET_TTL_EDGE_LEVEL

Function: Selects whether a TTL input is edge-sensitive or level sensitive.
Input: IPPARIO_TTL_BITS structure
Output: None
Notes: Determines whether the interrupt for each of the enabled TTL input lines responds to a static logic level or a transition between levels (1 = edge, 0 = level). The number of valid bits varies with the number of TTL I/O. This call is not valid on the IP-Parallel-485 board as it has no TTL drivers.

IOCTL_IPPAR??_GET_TTL_EDGE_LEVEL

Function: Returns the interrupt edge/level values set in the previous call.
Input: None
Output: IPPARIO_TTL_BITS structure
Notes: The number of valid bits varies with the number of TTL I/O lines. This call is not valid on the IP-Parallel-485 board as it has no TTL drivers.

IOCTL_IPPAR??_SET_485_EDGE_LEVEL

Function: Selects whether an RS485 input is edge or level sensitive.
Input: unsigned long int
Output: None
Notes: Determines whether the interrupt for each of the enabled RS485 input lines will respond to a static logic level or a transition between levels (1 = edge, 0 = level). The number of valid bits varies with the number of RS485 I/O. This call is not valid on the IP-Parallel-TTL board as it has no RS485 drivers.

IOCTL_IPPAR??_GET_485_EDGE_LEVEL

Function: Returns the interrupt edge/level values set in the previous call.
Input: None
Output: unsigned long int
Notes: The number of valid bits varies with the number of RS485 I/O. This call is not valid on the IP-Parallel-TTL board as it has no RS485 drivers.

IOCTL_IPPAR??_SET_TTL_POLARITY

Function: Selects whether a TTL input is active high or active low.
Input: IPPARIO_TTL_BITS structure
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). The number of valid bits varies with the number of TTL I/O. This call is not valid on the IP-Parallel-485 board as it has no TTL drivers.

IOCTL_IPPAR??_GET_TTL_POLARITY

Function: Returns the interrupt polarity values set in the previous call.
Input: None
Output: IPPARIO_TTL_BITS structure
Notes: The number of valid bits varies with the number of TTL I/O lines. This call is not valid on the IP-Parallel-485 board as it has no TTL drivers.

IOCTL_IPPAR??_SET_485_POLARITY

Function: Selects whether an RS485 input is active high or active low.
Input: unsigned long int
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). The number of valid bits varies with the number of RS485 I/O. This call is not valid on the IP-Parallel-TTL board as it has no RS485 drivers.

IOCTL_IPPAR??_GET_485_POLARITY

Function: Returns the interrupt polarity values set in the previous call.
Input: None
Output: unsigned long int
Notes: The number of valid bits varies with the number of RS485 I/O. This call is not valid on the IP-Parallel-TTL board as it has no RS485 drivers.

IOCTL_IPPAR??_READ_DIRECT

Function: Reads the input data bus directly.
Input: None
Output: IPPARIO_READ_DATA structure
Notes: This call reads the raw real-time input data from the TTL and RS485 input lines and returns an. IPPARIO_READ_DATA structure. This structure has separate fields for the TTL and RS485 data bits. The number of valid bits in each field varies with the number of each type of I/O.

IOCTL_IPPAR??_READ_ FILTERED

Function: Reads the filtered input data.
Input: None
Output: IPPARIO_READ_DATA structure
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 values are returned in a IPPARIO_READ_DATA structure, which has separate fields for the TTL and RS485 data bits. The number of valid bits in each field varies with the number of each type of I/O. The latched bits are automatically cleared when read by this call.

IOCTL_IPPAR??_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_IPPAR??_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_IPPAR??_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_IPPAR??_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_IPPAR??_LOAD_COUNTER

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

IOCTL_IPPAR??_CLEAR_TIMER

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

IOCTL_IPPAR??_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.

IOCTL_IPPAR??_GET_STATUS

Function: Returns the status bits in the INT_STAT register.
Input: None
Output: unsigned short int
Notes: The interrupt status bits are read by this call and the latched bits are then automatically cleared. See DDIpPar??.h for the bit definitions.

IOCTL_IPPAR??_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_IPPAR??_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. Also checks the state of the IP slot control register interrupt 0 enable bit in the saved configuration, and sets it if needed. 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_IPPAR??_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.

IOCTL_IPPAR??_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_IPPAR??_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_IPPAR??_GET_VECTOR

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

IOCTL_IPPAR??_GET_ISR_STATUS

Function: Returns the interrupt status and vector read in the last ISR.
Input: None
Output: IPPAR??_INT_STAT structure
Notes: The status contains the contents of the INT_STAT register read in the last ISR execution. Also, if bit 12 is set, it indicates that a bus error occurred for this IP slot.

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 Street, Suite C
Santa Cruz, CA 95060
831-457-8891
831-457-4793 Fax
support@dyneng.com