DYNAMIC ENGINEERING
150 Dubois St. Ste 3, Santa Cruz, Calif. 95060
831-457-8891 **Fax** 831-457-4793
http://www.dyneng.com
sales@dyneng.com
Est. 1988

Driver Documentation

IP-Pulse

4 Channel Digital Pulse Generator
IP Module

–TTL	:	4 TTL / 0 422
–1	:	3 TTL / 1 422
–2	:	2 TTL / 2 422
–3	:	1 TTL / 3 422
–422	:	0 TTL / 4 422

Win32 Driver Model

Revision A1
Corresponding Hardware: Revision B
PROM Revision D
10-2001-0102

IpPulse
WDM Device Driver for the
IP-Pulse IP Modules

Dynamic Engineering
435 Park Drive
Ben Lomond, CA 95005
831- 457-8891
831-457-4793 FAX

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_IPPULSE_GET_INFO 9
IOCTL_IPPULSE_SET_IP_CONTROL 9
IOCTL_IPPULSE_GET_IP_CONTROL 9
IOCTL_IPPULSE_SET_PULSE_PARAMS 9
IOCTL_IPPULSE_GET_PULSE_PARAMS 10
IOCTL_IPPULSE_SET_PULSE_CONFIG 10
IOCTL_IPPULSE_GET_PULSE_CONFIG 10
IOCTL_IPPULSE_ENABLE_PULSE 10
IOCTL_IPPULSE_DISABLE_PULSE 10
IOCTL_IPPULSE_GET_INT_STATUS 11
IOCTL_IPPULSE_REGISTER_EVENT 11
IOCTL_IPPULSE_ENABLE_INTERRUPT 11
IOCTL_IPPULSE_DISABLE_INTERRUPT 11
IOCTL_IPPULSE_FORCE_INTERRUPT 12
IOCTL_IPPULSE_SET_VECTOR 12
IOCTL_IPPULSE_GET_VECTOR 12
IOCTL_IPPULSE_GET_ISR_STATUS 12

WARRANTY AND REPAIR **13**

Service Policy **13**
Out of Warranty Repairs 13

For Service Contact: **13**

Introduction

Note: In this document IpPulse refers to the IpPulse driver that is designed to be the software interface for one of five versions of the IP-Pulse IP module. The versions and the I/O distributions are as follows:

Board Type	IO configuration
IP-Pulse-TTL	4 TTL/CMOS pulse generators
IP-Pulse-1	3 TTL/CMOS, 1 differential pulse generators
IP-Pulse-2	2 TTL/CMOS, 2 differential pulse generators
IP-Pulse-3	1 TTL/CMOS, 3 differential pulse generators
IP-Pulse-422	4 differential pulse generators

The IpPulse driver is a Win32 driver model (WDM) device driver for the IP-Pulse board from Dynamic Engineering. Each IP- Pulse board implements four independent pulse generators using I/O driver standards of either TTL/CMOS, RS422 or a combination of both. A separate Device Object controls each IP-Pulse board, and a separate handle references each Device Object. IO Control calls (IOCTLs) are used to configure and control 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-Pulse 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 distribution. These files include IpPulse.sys, IpDevice.inf, DDIpPulse.h, IpPulseGUID.h, IpPulseDef.h, IpPulseTest.exe, and IpPulseTest source files.

Windows 2000 Installation

Copy IpDevice.inf and IpPulse.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 to the WINDOWS\INF folder and copy IpPulse.sys to a floppy disk, or CD if preferred. Right click on the IpDevice.inf file icon in the WINDOWS\INF folder and select ***Install*** from the pop-up menu. This will create a precompiled information file (.pnf) in the same directory.

With the hardware installed, power-on the PCI host computer and wait for the ***Found New Hardware Wizard*** dialogue window to appear. The **IP-Pulse** should be named in the dialogue box. Follow the steps below:

• Insert the disk prepared above in the appropriate drive.
• Select ***Install from a list or specific location***
• Select ***Next***
• Select ***Don't search. I will choose the driver to install***
• Select ***Next***
• Select ***Show all devices*** from the list
• Select ***Next***
• Select ***Dynamic Engineering*** from the Manufacturer list
• Select ***IP-Pulse Device*** from the Model list
• Select ***Next***
• Select ***Yes*** on the Update Driver Warning dialogue box.
• Enter the drive, *e.g.* ***A:\***, in the ***Files Needed*** dialogue box.
• Select ***OK***.
• Select ***Finish*** to close the ***Found New Hardware Wizard***.

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

The DDIpPulse.h file is the C header file that defines the Application Program Interface (API) to the driver. The IpPulseGUID.h file is a C header file that defines the device interface identifier for the IpPulse. These files are required at compile time by any application that wishes to interface with the IpPulse driver. The IpPulseDef.h file contains the relevant bit defines for the IpP-Pulse registers. These files are not needed for driver installation.

The IpPulseTest.exe file is a sample Win32 console application that makes calls into the IpPulse 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 ***IpPulseTest –d0 -?*** to display a list of commands (the IpPulseTest.exe file must be in the directory that the window is referencing). All of the commands are of the form ***IpPulseTest –d<u>n</u> –i<u>m</u>*** where ***<u>n</u>*** and ***<u>m</u>*** 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.

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 IpPulseGUID.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 instance of an interface
#define MAX_DEVICE_NAME 256
// Handle to the device object
HANDLE                          hIpPulse = 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_IPPULSE,
                                  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);

// Find the interface for device 0
if(!SetupDiEnumDeviceInterfaces(hDeviceInfo,
                                NULL,
                                (LPGUID)&GUID_DEVINTERFACE_IPPULSE,
                                0,
                                &interfaceData))
{
   status = GetLastError();
   if(status == ERROR_NO_MORE_ITEMS)
   {
      printf("**Error: couldn't find device(no more items), (%d)\n", 0);
      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
hIpPulse = CreateFile(deviceName,
                      GENERIC_READ    | GENERIC_WRITE,
                      FILE_SHARE_READ | FILE_SHARE_WRITE,
                      NULL,
                      OPEN_EXISTING,
                      NULL,
                      NULL);

if(hIpPulse == 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 and control the device. IOCTLs refer to a single Device Object in the driver, which controls a single board. IOCTLs are called using the Win32 function DeviceIoControl(), and passing in the handle to the device opened with CreateFile(). IOCTLs generally have input parameters, output parameters, or both. Often a custom structure is used.

IOCTL_IPPULSE_GET_INFO

Function: Returns the current driver version and instance number.
Input: none
Output: DRIVER_IPPULSE_DEVICE_INFO structure
Notes: This call does not access the hardware, only driver parameters. See DDIpPulse.h for the definition of DEVICE_IPPULSE_DEVICE_INFO.

IOCTL_IPPULSE_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 IpPulseDef.h header file for more information.

IOCTL_IPPULSE_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 IpPulseDef.h header file for more information.

IOCTL_IPPULSE_SET_PULSE_PARAMS

Function: Configure pulse, timing parameters for one of four channels.
Input: PULSE_PARAM structure
Output: none
Notes: Controls the pulse on time, off time, count, and shift values for the specified channel. Refer to DDIpPulse.h for the definition of the PULSE_PARAM structure.

IOCTL_IPPULSE_GET_PULSE_PARAMS

Function: Returns the pulse timing parameters for the specified channel.
Input: unsigned character (channel number)
Output: PULSE_PARAM structure
Notes: Returns the values set in the previous call.

IOCTL_IPPULSE_SET_PULSE_CONFIG

Function: Controls various details of a pulse channel's configuration.
Input: PULSE_CONFIG structure
Output: none
Notes: Set or clear bits for PulseEnable, IntEnable, IntEachPulse, InvertPulse, OnTimeEnable, OffTimeEnable, and ShiftEnable for a single pulse channel. The last three values cause the respective values to be applied to the specified channel's pulse; the values applied are set in IOCTL_IPPULSE_SET_PULSE_PARAMS. Refer to DDIpPulse.h for the definition of the PULSE_CONFIG structure.

IOCTL_IPPULSE_GET_PULSE_CONFIG

Function: Returns the pulse configuration of the channel specified.
Input: unsigned character (channel number)
Output: PULSE_CONFIG structure
Notes: Returns the values set in the previous call.

IOCTL_IPPULSE_ENABLE_PULSE

Function: Master pulse Enable.
Input: None
Output: None
Notes: Pulse outputs on all channels are enabled to operate based on local enable and channel control registers.

IOCTL_IPPULSE_DISABLE_PULSE

Function: Master pulse Disable.
Input: None
Output: None
Notes: Terminates pulse outputs for all channels on next "off" phase.

IOCTL_IPPULSE_GET_INT_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 IpPulseDef.h for the bit definitions.

IOCTL_IPPULSE_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_IPPULSE_ENABLE_INTERRUPT

Function: Enables the master interrupt.
Input: none
Output: none
Notes: Sets the master interrupt enable, leaving all other bit values in the IPPULSE_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_IPPULSE_DISABLE_INTERRUPT

Function: Disables the master interrupt.
Input: none
Output: none
Notes: Clears the master interrupt enable, leaving all other bit values in the IPPULSE_BASE register unchanged. This IOCTL is used when interrupt processing is no longer desired.

IOCTL_IPPULSE_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_IPPULSE_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_IPPULSE_GET_VECTOR

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

IOCTL_IPPULSE_GET_ISR_STATUS

Function: Returns the interrupt status and vector read in the last ISR.
Input: none
Output: IPPULSE_INT_STAT structure
Notes: The status contains the interrupt vector and the contents of the INT_STAT register read in the last ISR execution. Also, if bit 12 is set in the interrupt status, 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 St. Ste. 3
Santa Cruz, CA 95060
831-457-8891
831-457-4793 fax
support@dyneng.com