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

Windows 10 WDF Driver Documentation

Developed with Windows Driver Foundation
Ver1.19

Revision A
Corresponding Hardware: Revision B
10-2009-0203
IpPulse
WDF Device Driver for the IP-Pulse IP Modules

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.


Trademarks and registered trademarks are owned by their respective manufactures.

Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>4</td>
</tr>
<tr>
<td>Note</td>
<td>5</td>
</tr>
<tr>
<td>Driver Installation</td>
<td>6</td>
</tr>
<tr>
<td>Windows 10 Installation</td>
<td>6</td>
</tr>
<tr>
<td>Driver Startup</td>
<td>7</td>
</tr>
<tr>
<td>IO Controls</td>
<td>7</td>
</tr>
<tr>
<td>IOCTL_IP_PULSE_GET_INFO</td>
<td>8</td>
</tr>
<tr>
<td>IOCTL_IP_PULSE_SET_IP_CONTROL</td>
<td>8</td>
</tr>
<tr>
<td>IOCTL_IP_PULSE_GET_IP_STATE</td>
<td>9</td>
</tr>
<tr>
<td>IOCTL_IP_PULSE_SET_PULSE_PARAMS</td>
<td>9</td>
</tr>
<tr>
<td>IOCTL_IP_PULSE_GET_PULSE_PARAMS</td>
<td>11</td>
</tr>
<tr>
<td>IOCTL_IP_PULSE_SET_PULSE_CONFIG</td>
<td>11</td>
</tr>
<tr>
<td>IOCTL_IP_PULSE_GET_PULSE_CONFIG</td>
<td>11</td>
</tr>
<tr>
<td>IOCTL_IP_PULSE_ENABLE_PULSE</td>
<td>12</td>
</tr>
<tr>
<td>IOCTL_IP_PULSE_DISABLE_PULSE</td>
<td>12</td>
</tr>
<tr>
<td>IOCTL_IP_PULSE_GET_INT_STATUS</td>
<td>12</td>
</tr>
<tr>
<td>IOCTL_IP_PULSE_REGISTER_EVENT</td>
<td>12</td>
</tr>
<tr>
<td>IOCTL_IP_PULSE_ENABLE_INTERRUPT</td>
<td>13</td>
</tr>
<tr>
<td>IOCTL_IP_PULSE_DISABLE_INTERRUPT</td>
<td>13</td>
</tr>
<tr>
<td>IOCTL_IP_PULSE_FORCE_INTERRUPT</td>
<td>13</td>
</tr>
<tr>
<td>IOCTL_IP_PULSE_SET_VECTOR</td>
<td>13</td>
</tr>
<tr>
<td>IOCTL_IP_PULSE_GET_VECTOR</td>
<td>13</td>
</tr>
<tr>
<td>IOCTL_IP_PULSE_GET_ISR_STATUS</td>
<td>14</td>
</tr>
</tbody>
</table>

**WARRANTY AND REPAIR**  15

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service Policy</td>
<td>15</td>
</tr>
<tr>
<td>Support</td>
<td>15</td>
</tr>
</tbody>
</table>

For Service Contact:  15
Introduction

The IP-Pulse driver is a Windows device driver for the IP- Industry-pack (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). Each IP- Pulse board implements four independent pulse generators using I/O driver standards of either TTL/CMOS, RS422 or a combination of both

The IP-Pulse driver package has two parts. The driver is installed into the Windows® OS, and the User Application “UserApp” executable.

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

UserApp 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 IpPulseUserApp also included several menu items that can be used as is to operate the IP Pulse. Refer to the IP Pulse User App Quick guide for details on how this can be done.

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.

**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:

<table>
<thead>
<tr>
<th>Board Type</th>
<th>IO configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP-Pulse-TTL</td>
<td>4 TTL/CMOS pulse generators</td>
</tr>
<tr>
<td>IP-Pulse-1</td>
<td>3 TTL/CMOS, 1 differential pulse generators</td>
</tr>
<tr>
<td>IP-Pulse-2</td>
<td>2 TTL/CMOS, 2 differential pulse generators</td>
</tr>
<tr>
<td>IP-Pulse-3</td>
<td>1 TTL/CMOS, 3 differential pulse generators</td>
</tr>
<tr>
<td>IP-Pulse-422</td>
<td>4 differential pulse generators</td>
</tr>
</tbody>
</table>

**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

There are several files provided in each driver package. These files include IpPulse.sys, IpPulsePublic.h, IpPublic.h, IpPulse.inf and Ippulse.cat.

IpPulsePublic.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.

Note: Other IP module drivers are included in the package since they were all signed together and must be present to validate the digital signature. These other IP module driver files must be present when the IpPulse driver is installed, to verify the digital signature in Ippulse.cat, otherwise they can be ignored.

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

Windows 10 Installation

Copy IpPulse.inf, Ippulse.cat, and IpPulse.sys 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 IpPulse 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.

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 IpPulsePublic.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 IpPulse device.

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.

```c
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 IpPulse driver are described below:

**IOCTL_IP_PULSE_GET_INFO**

*Function:* Returns the current driver version and instance 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, design revision and other information
typedef struct _DRIVER_CARRIERDEVICE_INFO {
    UCHAR    DriverRev; // Driver revision
    UCHAR    FirmwareRev; // Firmware major revision
    UCHAR    FirmwareRevMin;// Firmware minor revision
    UCHAR    CPLDRev; //**Used for PCIe carriers only** 0xFF for others
    UCHAR    CPLDRevMin; //**Used for PCIe carriers only** 0xFF for others
    UCHAR    InstanceNum; // Zero-based device number
    UCHAR    SwitchValue; // Eight-bit user switch
    UCHAR    NumIpSlots; // Maximum number of IPs on the carrier
} DRIVER_CARRIERDEVICE_INFO, *PDRIVER_CARRIERDEVICE_INFO;
```

**IOCTL_IP_PULSE_SET_IP_CONTROL**

*Function:* Sets the configuration of the IP slot.

*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;
    BOOLEAN RdWordSel;
    BOOLEAN BsErrTmOutSel;
    BOOLEAN ActCountEn;
} IP_SLOT_CONTROL, *PIP_SLOT_CONTROL;
```
IOCTL_IP_PULSE_GET_IP_STATE

Function: Returns the configuration of the IP slot.
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.

```c
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_PULSE_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. See the definition of the PULSE_PARAM structure below.

```c
typedef struct _PULSE_PARAM {
    UCHAR    Channel;
    ULONG    OnTime;
    ULONG    OffTime;
    ULONG    Count;
    ULONG    Shift;
} PULSE_PARAM;
```
} PULSE_PARAM, *PPULSE_PARAM;
**IOCTL_IP_PULSE_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. See the definition of the PULSE_PARAM structure above.

**IOCTL_IP_PULSE_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_IP_PULSE_SET_PULSE_PARAMS. See the definition of the PULSE_CONFIG structure below.

```c
typedef struct _PULSE_CONFIG
{
    UCHAR Channel;
    BOOLEAN PulseEnable;
    BOOLEAN IntEnable;
    BOOLEAN IntEachPulse;
    BOOLEAN OnTimeEnable;
    BOOLEAN OffTimeEnable;
    BOOLEAN ShiftEnable;
    BOOLEAN InvertPulse;
} PULSE_CONFIG, *PPULSE_CONFIG;
```

**IOCTL_IP_PULSE_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. See the definition of the PULSE_CONFIG structure above.
**IOCTL_IP_PULSE_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_IP_PULSE_DISABLE_PULSE**

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

**IOCTL_IP_PULSE_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.

**IOCTL_IP_PULSE_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_IP_PULSE_ENABLE_INTERRUPT**

*Function:* Enables the master interrupt.
*Input:* none
*Output:* none
*Notes:* Sets the master interrupt enable, leaving all other bit values in the 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 ISR. This allows the driver to set the master interrupt enable without knowing the state of the other base configuration bits.

**IOCTL_IP_PULSE_DISABLE_INTERRUPT**

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

**IOCTL_IP_PULSE_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_PULSE_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_PULSE_GET_VECTOR**

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

**IOCTL_IP_PULSE_GET_ISR_STATUS**

*Function:* Returns the interrupt status and vector read in the last ISR.

*Input:* none

*Output:* IP_PULSE_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.

```c
typedef struct _IP_PULSE_INT_STAT
{
    USHORT   InterruptStatus;
    USHORT   InterruptVector;
} IP_PULSE_INT_STAT, *PIP_PULSE_INT_STAT;
```
Warranty and Repair

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

All information provided is Copyright Dynamic Engineering