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

IpTest

WDF Driver Documentation For the IP-Test module

Developed with Windows Driver Foundation Ver1.9

Manual Revision: A1 7/22/16
Corresponding PCB: 10-2001-0303/4
FLASH Revision: A

IpTest WDF Device Driver for the IP-HV-Test IP Module

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

Table of Contents

Introduction

The IpTest driver is a Windows device driver for the IP-Test Industry-pack (IP) module from Dynamic Engineering. This driver was developed with the Windows Driver Foundation version 1.9 (WDF) from Microsoft, specifically the Kernel-Mode Driver Framework (KMDF).

The IP-HV-TEST module is used to test IP carriers developed by Dynamic Engineering. IP-HV-TEST is an orderable configuration of IP-Parallel-HV. When combined with the carrier driver IO, INT, ID and MEM space access requests are supported including up to 64-bit MEM space accesses when running a 64-bit OS and installed in a carrier that supports 64-bit accesses.

When the IP carrier driver is started it will enumerate the carrierôs IP bus by reading the ID proms of installed IPs. If the IP module driver has been previously installed, it will be loaded and a Device Object will be created for each installed IP. A separate handle to each IP module can be obtained using CreateFile() calls. IO Control calls (IOCTLs) are used to configure the IP module and read the moduleôs status. WriteFile() and ReadFile() calls are used to move blocks of data in and out of the IP module.

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

Driver Installation

There are several files provided in each driver package. These files include IpTest.sys, IpTestPublic.h, IpPublic.h, WdfCoInstaller01009.dll, IpDevices.inf and IpDevices.cat.

IpTestPublic.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 IpTest driver is installed, to verify the digital signature in IpDevices.cat, otherwise they can be ignored.

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

Windows 7 Installation

Copy IpDevices.inf, IpDevices.cat, WdfCoInstaller01009.dll, IpTest.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 IpTest 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 IpTestPublic.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 IpTest 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.

```
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 IpTest driver are described below:

IOCTL_IP_TEST_GET_INFO

Function: Returns the driver and firmware revisions, module instance number and location and other information.
Input: None
Output: DRIVER_IP_DEVICE_INFO structure
Notes: This call does not access the hardware, only stored driver parameters. NewIpCntl indicates that the moduleȏs 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 {
   USHORT   DriverRev;
   USHORT   FirmwareRev;
   USHORT   InstanceNum;
   UCHAR    CarrierSwitch;  // 0..0xFF
   UCHAR    CarrierSlotNum; // 0..7 -> IP slots A, B, C, D, E, F, G or H
   BOOLEAN  NewIpCntl;      // New IP slot control bits
   WCHAR    LocationString[IP_LOC_STRING_SIZE];
} DRIVER_IP_DEVICE_INFO, *PDRIVER_IP_DEVICE_INFO;
```

IOCTL_IP_TEST_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_TEST_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_TEST_SET_BASE_CONFIG

Function: Sets configuration parameters in the IP base control register.
Input: IP_TEST_BASE_CONFIG structure
Output: None
Notes: Controls the output data latch behavior. 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 OUT_SEL and IP_TEST_BASE_CONFIG below.

```
typedef enum _OUT_SEL {
   DISABLE,
   ENABLE,
   AUTO
} OUT_SEL, *POUT_SEL;

 // Output control
typedef struct _IP_TEST_BASE_CONFIG {
   OUT_SEL  Outen;
} IP_TEST_BASE_CONFIG, *PIP_TEST_BASE_CONFIG;
```

IOCTL_IP_TEST_GET_BASE_CONFIG

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

IOCTL_IP_TEST_SET_TTL_DATA

Function: Sets the value of the 24 TTL data outputs on the board.
Input: Unsigned long integer
Output: None
Notes: The value of each of the 24 TTL data output bits is determined by the corresponding bit in the input word for this call. The TTL lines of the IP-Test module are equipped with open drain drivers with pull-ups to achieve a TTL high voltage level. When the drivers are configured to output a low level the external line will be driven low, but when configured to output a high level, the value of the external line will follow the level of the externally connected node.

IOCTL_IP_TEST_GET_TTL_DATA

Function: Returns the state of the bits in the output data register.
Input: None
Output: Unsigned long integer
Notes: The value returned depends only on the value that was written by IOCTL_IP_TEST_SET_TTL_DATA, not on the voltage level of the external line. To return the line voltage levels use IOCTL_IP_TEST_READ_DIRECT.

IOCTL_IP_TEST_SET_TTL_INT_EN

Function: Selects which TTL inputs are possibly latched and can cause an interrupt.
Input: Unsigned long integer
Output: None
Notes: This call defines the mask of which of the 24 TTL input lines will be enabled to cause an interrupt if the specified polarity and edge/level conditions are met. A one in a certain bit positions enables the respective input line to be considered. A zero eliminates that line as a candidate for interrupt generation.

IOCTL_IP_TEST_GET_TTL_INT_EN

Function: Returns the interrupt enable values set in the previous call.
Input: None
Output: Unsigned long integer
Notes: The value returned will be equal to the value that was written by the last IOCTL_IP_TEST_SET_TTL_INT_EN call.

IOCTL_IP_TEST_SET_TTL_EDGE_LEVEL

Function: Selects whether a TTL input is edge-sensitive or level sensitive.
Input: Unsigned long integer
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. A one in a certain bit positions configures the respective input line to be edge-sensitive. A zero configures the line to be level-sensitive. Which level or edge the latch responds to is determined by the IOCTL_IP_TEST_SET_TTL_POLARITY call.

IOCTL_IP_TEST_GET_TTL_EDGE_LEVEL

Function: Returns the interrupt edge/level values set in the previous call.
Input: None
Output: Unsigned long integer
Notes: The value returned will be equal to the value that was written by the last IOCTL_IP_TEST_SET_TTL_EDGE_LEVEL call.

IOCTL_IP_TEST_SET_TTL_POLARITY

Function: Selects whether a TTL input is active high or active low.
Input: Unsigned long integer
Output: None
Notes: Determines the polarity of the level or edge to which the corresponding input line will respond. A one in a certain bit positions configures the respective input line to respond to a high level or a rising edge depending on the state of the corresponding edge/level bit. A zero configures the line to respond to a low level or a falling edge.

IOCTL_IP_TEST_GET_TTL_POLARITY

Function: Returns the interrupt polarity values set in the previous call.
Input: None
Output: Unsigned long integer
Notes: The value returned will be equal to the value that was written by the last IOCTL_IP_TEST_SET_TTL_POLARITY call.

IOCTL_IP_TEST_READ_DIRECT

Function: Reads the input data bus directly.
Input: None
Output: Unsigned long integer
Notes: This call reads the raw real-time input data from the TTL input lines and returns an unsigned long integer corresponding to that value.

IOCTL_IP_TEST_READ_FILTERED

Function: Reads the state of the input data latches.
Input: None
Output: Unsigned long integer
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 an unsigned long integer. The latched bits are automatically cleared after being read by this call.

IOCTL_IP_TEST_SET_WR_MEM_OFFSET

Function: Specifies the starting offset into the 4K byte RAM for multi-word writes.
Input: Unsigned long integer
Output: None
Notes: This call should be run before a WriteFile() is performed the first time or if the target RAM address has changed since the last time this call was made.

IOCTL_IP_TEST_SET_RD_MEM_OFFSET

Function: Specifies the starting offset into the 4K byte RAM for multi-word reads.
Input: Unsigned long integer
Output: None
Notes: This call should be run before a ReadFile() is performed the first time or if the target RAM address has changed since the last time this call was made.

IOCTL_IP_TEST_GET_MEM_ADDRESS

Function: Returns the state of the 22 IP module address lines when the last MEM select occurred.
Input: None
Output: Unsigned long integer
Notes: When the 8 Mbyte IP MEM space is accessed, the lower six address bits are specified with the six IP address lines and the remaining upper 16 address bits are written to the 16 IP data bits. The address bits are concatenated into a 22-bit address that accesses the 4 Mbyte by 16-bit IP MEM space. The byte selects specify which byte(s) are accessed. The 22-bit address is returned by this call for test purposes.

IOCTL_IP_TEST_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_TEST_ENABLE_INTERRUPT

Function: Sets the master interrupt enable.
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_TEST_DISABLE_INTERRUPT

Function: Clears the master interrupt enable.
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_TEST_FORCE_INTERRUPT

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

```
typedef struct _IP_TEST_INT_SEL {
   BOOLEAN  IntSet0;
   BOOLEAN  IntSet1;
} IP_TEST_INT_SEL, *PIP_TEST_INT_SEL;
```

IOCTL_IP_TEST_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. This call accesses the register where that vector is stored.

IOCTL_IP_TEST_GET_VECTOR

Function: Returns the current interrupt vector value.
Input: None
Output: Unsigned character
Notes: This call accesses the register where the interrupt vector is stored so that the vector value can be verified without generating an interrupt.

IOCTL_IP_TEST_GET_ISR_STATUS

Function: Returns the interrupt status and vector read in the last ISR.
Input: None
Output: IP_TEST_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.

```
 // Interrupt status and vector
typedef struct _IP_TEST_INT_STAT {
   USHORT   InterruptStatus;
   USHORT   InterruptVector;
} IP_TEST_INT_STAT, *PIP_TEST_INT_STAT;
```

IOCTL_IP_TEST_PUT_MEM_DATA

Function: Writes a single 16-bit word to a specific address offset in the IP MEM space.
Input: IP_TEST_DATA_WRITE structure
Output: None
Notes: The address can be any value below 4,194,304 (0x400000) and the data can be any 16-bit value. This address covers the entire 8 Mbyte IP MEM space and although there is only a 4 Kbyte RAM implemented, the call succeeds for the entire IP MEM space as if the entire 8 Mbyte memory was populated.

```
typedef struct _IP_TEST_DATA_WRITE {
   ULONG    Address;
   USHORT   Data;
} IP_TEST_DATA_WRITE, *PIP_TEST_DATA_WRITE;
```

IOCTL_IP_TEST_GET_MEM_DATA

Function: Returns the 16-bit data word read at the address offset.
Input: Address offset (unsigned long integer)
Output: Data (unsigned short integer)
Notes: This call reads the MEM space location at the input address offset and returns the 16-bit data word. Reads above the 4 Kbyte address limit of the RAM that is actually implemented will result in a bus error.

Write

Data can be written to the RAM using a WriteFile() call. The user supplies the device handle, a pointer to the buffer containing the data, the number of bytes to write, a pointer to a variable to store the amount of data actually transferred, and a pointer to an optional Overlapped structure for performing asynchronous IO. The number of bytes requested and the current write memory offset are checked to see how much data can be written without overflowing the memory. The command is executed with successive writes to the RAM. If the IP carrier is PCIe based and supports 64-bit writes, four successive 16-bit writes will be made to the IP for each 64-bit PCIe access. If the carrier is PCI based two16-bit writes will be made to the IP for each 32-bit PCI access.

Read

Data can be read from the RAM using a ReadFile() call. The user supplies the device handle, a pointer to the buffer that will contain the data, the number of bytes to read, a pointer to a variable to store the amount of data actually transferred, and a pointer to an optional Overlapped structure for performing asynchronous IO. The number of bytes requested and the current read memory offset are checked to see how much data can be read without going over the memory limit. The command is executed with successive reads from the RAM. If the IP carrier is PCIe based and supports 64-bit reads, four successive 16-bit reads will be made from the IP for each 64-bit PCIe access. If the carrier is PCI based two16-bit reads will be made from the IP for each 32-bit PCI access.

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 merchantability 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.

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