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

IP-Parallel-HaveQuick

“IpHQT”

Windows 10 WDF Driver Documentation

Developed with Windows Driver Foundation Ver1.9

Manual Revision 1p2
Corresponding Hardware: Revision 04
10-2001-0304
FLASH revision 0103

IpHQT

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

Table of Contents

Introduction

The IpHQT driver is a Windows device driver for IP-Parallel-HaveQuick 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 IpHQT software package has two parts. The driver for Windows® 10 OS, and the User Application “UserAp” executable.

The driver is delivered electronically. The files supplied are installed into the client system to allow access to the hardware. 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 tests to your application to get started. The tests are 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. If you have suggestions for extended features, special calls for particular set-ups or whatever please share them with us.

When the IpHQT board is recognized by the IP Carrier Driver, the carrier driver will start the IpHQT 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 IpHQT with the carrier switch setting and the slot number of the IpHQT device. From within the IpHQT driver the user can access the switch and slot information to determine the specific device being accessed when more than one is installed.

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.

Note

This documentation will provide information about all calls made to the drivers, and how the drivers interact with the device for each of these calls. For more detailed information on the hardware implementation, refer to the IpHQT user manual (also referred to as the hardware manual).

Driver Installation

There are several files provided in each IP driver package. These files include .sys, .cat, .inf.

Please note: Your carrier driver may need to be updated to use the IP module. The list of IP modules is compiled along with the driver and due to signing requirements.

Public.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. IpPublic.h is supplied with the carrier driver. Public.h. is supplied with UserAp.

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

Windows 10 Installation

Copy the supplied system files to a folder of your choice.

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 IpBis6Gpio 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 Public.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 IpHQT 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 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
```

IOCTLs defined for the IpHQT driver are described below:

IOCTL_IP_HQT_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 module's carrier has expanded slot control capabilities. See the definition of DRIVER_IP_DEVICE_INFO below.

```
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_HQT_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_HQT_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_HQT_WRITE_SYNC_WORD

Function: Sets 16-bit sync word.
Input: USHORT
Output: none
Notes: Default value for the sync word is 0x11E9

IOCTL_IP_HQT_READ_SYNC_WORD

Function: Reads 16-bit sync word.
Input: none
Output: USHORT
Notes: none

IOCTL_IP_HQT_LOAD_TX_TIME

Function: Loads time from the time registers in the transmit register.
Input: none
Output: none
Notes: When the time is written to the time registers, it is not automatically ready to transmit. The time must be loaded into the transmission registers before transmission can start.

IOCTL_IP_HQT_START_TX

Function: Starts transmission out of the device at the pulse intervals.
Input: none
Output: none
Notes: This is done by setting the TX 'run' bit in the TX Control register.

IOCTL_IP_HQT_STOP_TX

Function: Loads time from the time registers in the transmit register.
Input: none
Output: none
Notes: This is done by turning off the TX 'run' bit.

IOCTL_IP_HQT_INIT_SAMPLE_COUNT

Function: Reset sample count - sets 16-bit time tag value to zero.
Input: none
Output: none
Notes: This is done by setting the sample count clear bit in the RX Control register.

IOCTL_IP_HQT_START_RX

Function: Enable receiver to read time value transmission and write times into the FIFO.
Input: none
Output: none
Notes: This is done by setting the RX 'Start' bit in the RX Control register.

IOCTL_IP_HQT_STOP_RX

Function: Disables receiver to stop reading time values.
Input: none
Output: none
Notes: Stopping transmission is necessary to clear the "Real-time data valid" bit in the status register.

IOCTL_IP_HQT_READ_TIME

Function: Reads the time received from the RX receiver (Not from the FIFOs).
Input: none
Output: IP_HQT_READ_TIME_DATA
Notes: This is the time placed into the data registers (not the time registers used to transmit time and not the FIFOs). The data is the data currently stored in the registers. The act of reading the data advances the data in the 2 deep pipeline to the next value. This prevents the data from changing during the act of reading from the multiple registers. If not polling, read twice to get the current data set.

```
typedef struct _IP_HQT_READ_TIME_DATA {
    HAVE_QUICK_TIME   Time; //same structure used to set the time for transmission
    USHORT            SecFraction;
} IP_HQT_READ_TIME_DATA, * PIP_HQT_READ_TIME_DATA
```

IOCTL_IP_HQT_RESET_FIFO

Function: Resets FIFOs so they are empty.
Input: none
Output: none
Notes: None.

IOCTL_IP_HQT_READ_FIFO

Function: Resets FIFOs so they are empty.
Input: none
Output: IP_HQT_FIFO_READ_DATA
Notes: None.

```
typedef struct _IP_HQT_FIFO_READ_DATA {
    USHORT            SampleNum;
    HAVE_QUICK_TIME   Time;
} IP_HQT_FIFO_READ_DATA, * PIP_HQT_FIFO_READ_DATA;
```

IOCTL_IP_HQT_GET_STATUS

Function: Reads interrupt status register.
Input: none
Output: USHORT
Notes: none

IOCTL_IP_HQT_REGISTER_EVENT

Function: Registers an event based on the type(s) designated in the INT_EVENT values. There are separate events that can be registered based on the types of interrupts the software is waiting for. This is also used to clear events by writing NULL.
Input: INT_EVENT
Output: none
Notes: The caller creates an event with CreateEvent() and supplies the handle returned from that call as the input to this IOCTL along with the type of interrupt the software is waiting for. 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.

```
typedef enum _eventType {
    FORCE_INT,
    RX_INT,
    TX_INT,
    OTHER
} eventType;

typedef struct _INT_EVENT {
    HANDLE pIntEvent; //the handle returned from CreateEvent()
    eventType evtype;
}INT_EVENT, *PINT_EVENT;
```

IOCTL_IP_HQT_ENABLE_INTERRUPT

Function: Enables interrupts on both the carrier for this particular slot and enables the master interrupt for the IP Card.
Input: none
Output: none
Notes: none

IOCTL_IP_HQT_DISABLE_INTERRUPT

Function: Disables interrupts on both the carrier for this particular slot and enables the master interrupt for the IP Card.
Input: none
Output: none
Notes: none

IOCTL_IP_HQT_FORCE_INTERRUPT

Function: Enables the force interrupt bit to trigger an interrupt.
Input: none
Output: none
Notes: This will not cause an interrupt unless the interrupts were enabled with the IOCTL_IP_HQT_ENABLE_INTERRUPT. Further, this bit is disabled (turned off) in the Interrupt Service Routine (so if you check the bit after the interrupt occurs it will not show the bit as set).

IOCTL_IP_HQT_GET_INT_ENABLES

Function: Gets the int enable bit values from the RX Control and TX Control registers.
Input: none
Output: INT_ENABLES
Notes: Bit definitions can be found in the '_TX_CONTROL' and '_RX_CONTROL' sections under Register Definitions in the Hardware manual.

```
typedef struct _IP_HQT_INT_ENABLES {
    BOOLEAN     TxInten;
    BOOLEAN     RxInten;
    BOOLEAN     RxPerrInten;
    BOOLEAN     RxFerrInten;
} IP_HQT_INT_ENABLES, * PIP_HQT_INT_ENABLES;
```

IOCTL_IP_HQT_SET_INT_ENABLES

Function: Sets the int enable bits in the RX Control and TX Control registers based on the values passed in the INT_ENABLES structure.
Input: INT_ENABLES
Output: none
Notes: Bit definitions can be found in the '_TX_CONTROL' and '_RX_CONTROL' sections under Register Definitions in the Hardware manual.

```
typedef struct _IP_HQT_INT_ENABLES {
    BOOLEAN     TxInten;
    BOOLEAN     RxInten;
    BOOLEAN     RxPerrInten;
    BOOLEAN     RxFerrInten;
} IP_HQT_INT_ENABLES, * PIP_HQT_INT_ENABLES;
```

IOCTL_IP_HQT_READ_TIME_REGS

Function: Reads time values into the various time registers in the device, as well as the “time figure of merit” from the main control register.
Input: none
Output: HAVE_QUICK_TIME
Notes: Definitions of various time registers and the control register can be found in the in the Hardware manual.

```
typedef struct _HAVE_QUICK_TIME {
    UCHAR     Hour;
    UCHAR     Minute;
    UCHAR     Second;
    USHORT    DayOfYear;
    USHORT    Year;
    UCHAR     Tfom;
} HAVE_QUICK_TIME, * PHAVE_QUICK_TIME;
```

IOCTL_IP_HQT_READ_TIME_REGS

Function: Reads time values into the various time registers in the device, as well as the “time figure of merit” from the main control register.
Input: none
Output: HAVE_QUICK_TIME
Notes: Definitions of various time registers and the control register can be found in the in the Hardware manual.

```
typedef struct _HAVE_QUICK_TIME {
    UCHAR     Hour;
    UCHAR     Minute;
    UCHAR     Second;
    USHORT    DayOfYear;
    USHORT    Year;
    UCHAR     Tfom;
} HAVE_QUICK_TIME, * PHAVE_QUICK_TIME;
```

IOCTL_IP_HQT_WRITE_TIME_REGS

Function: Writes time values into the various time registers in the device, as well as the "time figure of merit" in the main control register.
Input: HAVE_QUICK_TIME
Output: none
Notes: Definitions of various time registers and the control register can be found in the in the Hardware manual.

```
//See UserApp for example of converting standard system time function to this format
typedef struct _HAVE_QUICK_TIME {
    UCHAR    Hour;
    UCHAR    Minute;
    UCHAR    Second;
    USHORT   DayOfYear;
    USHORT   Year;
    UCHAR    Tfom;
} HAVE_QUICK_TIME, * PHAVE_QUICK_TIME;
```

IOCTL_IP_HQT_SET_VECTOR

Function: Writes 16-bit value to the vector register.
Input: USHORT
Output: none
Notes: none

IOCTL_IP_HQT_GET_VECTOR

Function: Reads 16-bit value from the vector register.
Input: none
Output: USHORT
Notes: none

IOCTL_IP_HQT_GET_ISR_STATUS

Function: Reads 16-bit value to the vector register.
Input: none
Output: IP_HQT_INT_STAT
Notes: none

```
typedef struct _IP_HQT_INT_STAT {
    USHORT   InterruptStatus;
    USHORT   InterruptVector;
} IP_HQT_INT_STAT, * PIP_HQT_INT_STAT;
```

IOCTL_IP_HQT_GET_BASE_CONTROL

Function: Reads 16-bit value from the Base Control register of the IpHQT.
Input: USHORT
Output: none
Notes: none

IOCTL_IP_HQT_SET_BASE_CONTROL

Function: Writes 16-bit value to the Base Control register of the IpHQT.
Input: none
Output: USHORT
Notes: none

IOCTL_IP_HQT_SET_STATUS

Function: Writes 16-bit value to the Status register of the IpHQT.
Input: USHORT
Output: none
Notes: This can be used to turn of status bits if they are not turned off in the ISR (this may have happened if interrupts are triggered on the card, but not passed to the PCI bus).

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

The driver has gone through extensive testing, and while not infallible, problems experienced will likely be “cockpit error” rather than an error with the driver. We will work with you to determine the cause of the issue. If the effort is more than a quick conversation, we will offer a support contract. We can write updates to the driver to add features, create middleware etc.

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