

DYNAMIC ENGINEERING
435 Park Dr., Ben Lomond, Calif. 95005
831-336-8891 Fax 831-336-3840
<http://www.dyneng.com>
sales@dyneng.com
Est. 1988

IpTape

Driver Documentation

Win32 Driver Model

Revision B
Corresponding Hardware: Revision B
10-2001-0102
Corresponding Firmware: Revision A,B

IpTape WDM Device Driver for the IP-Tape IP Module

Dynamic Engineering
435 Park Drive
Ben Lomond, CA 95005
831- 336-8891
831-336-3840 FAX

©2004 by Dynamic Engineering.

Trademarks and registered trademarks are owned by their
respective manufactures.
Manual Revision B. Revised May 14, 2004.

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.



Table of Contents

Introduction	5
Note	5
Driver Installation	5
Windows 2000 Installation	5
Windows XP Installation	6
Driver Startup	7
IO Controls	10
IOCTL_IPTAPE_GET_INFO	10
IOCTL_IPTAPE_SET_IP_CONTROL	10
IOCTL_IPTAPE_GET_IP_CONTROL	10
IOCTL_IPTAPE_SET_PARITY	11
IOCTL_IPTAPE_GET_PARITY	11
IOCTL_IPTAPE_WRITE_MEM_WORD	11
IOCTL_IPTAPE_INIT_MEM_READ	11
IOCTL_IPTAPE_GET_MEM_DATA	11
IOCTL_IPTAPE_SET_CLOCK_CONFIG	12
IOCTL_IPTAPE_GET_CLOCK_CONFIG	12
IOCTL_IPTAPE_SET_INT_EN	12
IOCTL_IPTAPE_GET_INT_EN	12
IOCTL_IPTAPE_SET_EDGE_LEVEL	12
IOCTL_IPTAPE_GET_EDGE_LEVEL	13
IOCTL_IPTAPE_SET_POLARITY	13
IOCTL_IPTAPE_GET_POLARITY	13
IOCTL_IPTAPE_READ_DIRECT	13
IOCTL_IPTAPE_READ_FILTERED	13
IOCTL_IPTAPE_GET_STATUS	14
IOCTL_IPTAPE_REGISTER_EVENT	14
IOCTL_IPTAPE_ENABLE_INTERRUPT	14
IOCTL_IPTAPE_DISABLE_INTERRUPT	15
IOCTL_IPTAPE_FORCE_INTERRUPT	15
IOCTL_IPTAPE_SET_VECTOR	15
IOCTL_IPTAPE_GET_VECTOR	15
IOCTL_IPTAPE_GET_ISR_STATUS	15
IOCTL_IPTAPE_SET_OUT_DATA	16
IOCTL_IPTAPE_GET_OUT_DATA	16



WARRANTY AND REPAIR	17
Service Policy	17
Out of Warranty Repairs	17
For Service Contact:	18



Introduction

The IpTape driver is a Win32 driver model (WDM) device driver for the Ip-Tape (IP-Parallel-BA1) board from Dynamic Engineering. Each IP-Tape board implements a parallel interface to a customer specific tape unit. A separate Device Object controls each IP-Tape 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 tape unit via 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-Tape 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 IpTape.sys, IpDevice.inf, DDIpTape.h, IpTapeGUID.h, IpTapeDef.h, IPTTest.exe, and IPTTest source files.

Windows 2000 Installation

Copy IpDevice.inf and IpTape.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 IpTape.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-Tape 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-Tape 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 DDIpTape.h file is the C header file that defines the Application Program Interface (API) to the driver. The IpTapeGUID.h file is a C header file that defines the device interface identifier for the IpTape. These files are required at compile time by any application that wishes to interface with the IpTape driver. The IpTapeDef.h file contains the relevant bit defines for the IP-Tape registers. These files are not needed for driver installation.



The IPTTest.exe file is a sample Win32 console application that makes calls into the IpTape 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 *IPTTest -dO -?* to display a list of commands (the IPTTest.exe file must be in the directory that the window is referencing). The commands are all of the form *IPTTest -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.

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 IpTapeGUID.h.

Below is example code for opening a handle for device O. 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 hIpTape = 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_IPTAPE,
                                  NULL,
                                  NULL,
```



```

DIGCF_DEVICEINTERFACE);

DIGCF_PRESENT |

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

if(hIpTape == 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 and pass data in and out. 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_IPTAPE_GET_INFO

Function: Returns the current driver version.

Input: none

Output: DRIVER_IP_DEVICE_INFO structure

Notes: This call does not access the hardware, only driver parameters. See DDIpTape.h for the definition of DEVICE_IP_DEVICE_INFO.

IOCTL_IPTAPE_SET_IP_CONTROL

Function: Sets the configuration of the IP slot.

Input: ULONG

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 IpTapeDef.h header file for more information.

IOCTL_IPTAPE_GET_IP_CONTROL

Function: Returns the configuration of the IP slot.

Input: none

Output: ULONG

Notes: Returns the slot configuration register value. See the bit definitions in the IpTapeDef.h header file for more information.



IOCTL_IPTAPE_SET_PARITY

Function: Sets the Read and Write parity definitions.

Input: PARITY_DEF structure

Output: none

Notes: Controls whether odd or even parity is used in read and write transactions. See the bit structure definition in the DDlpTape.h header file for more information.

IOCTL_IPTAPE_GET_PARITY

Function: Returns the Read and Write parity definitions.

Input: none

Output: PARITY_DEF structure

Notes: Returns the read and write parity configuration.

IOCTL_IPTAPE_WRITE_MEM_WORD

Function: Writes one data word to the specified memory address.

Input: MEM_STRUCT structure

Output: none

Notes: The MEM_STRUCT structure has two fields, Address and Data. This call writes the Data contents to the specified Address as a background process.

IOCTL_IPTAPE_INIT_MEM_READ

Function: Starts a memory read cycle from the specified address.

Input: ULONG

Output: none

Notes: This call starts a read cycle from the specified address as a background process. Once the busy bit in the status register is cleared, the data read can be retrieved with the next IOCTL.

IOCTL_IPTAPE_GET_MEM_DATA

Function: Returns the data read in the previous memory read cycle.

Input: none

Output: USHORT

Notes: This call is used in conjunction with the previous call to read data from the tape unit.



IOCTL_IPTAPE_SET_CLOCK_CONFIG

Function: Writes a value to the clock control register.

Input: USHORT

Output: none

Notes: Controls the clock divisor, the input clock source, and whether the input clock or the divided clock is selected.

IOCTL_IPTAPE_GET_CLOCK_CONFIG

Function: Returns the value from the clock control register.

Input: none

Output: USHORT

Notes: Returns the clock divisor, the input clock source, and the output clock select control bits.

IOCTL_IPTAPE_SET_INT_EN

Function: Writes values to the interrupt enable registers.

Input: IO_BITS structure

Output: none

Notes: This call defines the mask of which of the 48 input lines will be enabled to cause an interrupt when the specified conditions are met (1 = enabled, 0 = disabled).

IOCTL_IPTAPE_GET_INT_EN

Function: Returns the values of the interrupt enable registers.

Input: none

Output: IO_BITS structure

Notes:

IOCTL_IPTAPE_SET_EDGE_LEVEL

Function: Writes values to the edge/level registers.

Input: IO_BITS structure

Output: none

Notes: Determines whether the interrupt for each of the input lines will respond to a static logic level or a transition between levels (1 = edge, 0 = level).



IOCTL_IPTAPE_GET_EDGE_LEVEL

Function: Returns the values from the edge/level registers.

Input: none

Output: IO_BITS structure

Notes:

IOCTL_IPTAPE_SET_POLARITY

Function: Writes values to the polarity registers.

Input: IO_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, 0 = non-inverted).

IOCTL_IPTAPE_GET_POLARITY

Function: Returns values from the polarity registers.

Input: none

Output: IO_BITS structure

Notes:

IOCTL_IPTAPE_READ_DIRECT

Function: Reads the direct input data.

Input: none

Output: IO_BITS structure

Notes: This call reads the raw real-time input data from the 48 TTL input lines.

IOCTL_IPTAPE_READ_FILTERED

Function: Reads the filtered input data registers.

Input: none

Output: IO_BITS 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. Reading these registers clears the latched bits.



IOCTL_IPTAPE_GET_STATUS

Function: Returns the status bits in the INT_STAT register.

Input: none

Output: USHORT

Notes: There are only two status bits in this register: Busy, indicating that the Tape interface is active with a read or write cycle, and Parity ok, indicating that the calculated parity of the last read cycle matched the stored parity.

IOCTL_IPTAPE_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_IPTAPE_ENABLE_INTERRUPT

Function: Enables the master interrupt.

Input: none

Output: none

Notes: Sets the master interrupt enable, leaving all other bit values in the IPTAPE_BASE register the same. Also checks the state of the IP slot control register interrupt O 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_IPTAPE_DISABLE_INTERRUPT

Function: Disables the master interrupt.

Input: none

Output: none

Notes: Clears the master interrupt enable, leaving all other bit values in the IPTAPE_BASE configuration register the same. This IOCTL is used when interrupt processing is no longer desired.

IOCTL_IPTAPE_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_IPTAPE_SET_VECTOR

Function: Sets the value of the interrupt vector.

Input: UCHAR

Output: none

Notes: This value is 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_IPTAPE_GET_VECTOR

Function: Returns the current interrupt vector value.

Input: none

Output: UCHAR

Notes:

IOCTL_IPTAPE_GET_ISR_STATUS

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

Input: none

Output: INT_STAT structure

Notes: The status contains the contents of the INT_STAT register read in the ISR. If bit 12 is set, it indicates that a bus error occurred for this IP slot.



IOCTL_IPTAPE_SET_OUT_DATA

Function: Writes a value to the TTL output data registers.

Input: IO_BITS structure

Output: none

Notes: This call can only be used with the 'B' or later revision of the IP-Tape PROM. It uses previously unused addresses to implement the IP-Parallel-TTL output data interface in order to allow loop-back testing of the IP-Tape without installing a -TTL PROM. If this call is executed on a board with a rev 'A' PROM, the call will succeed, but no data will be written to any register.

IOCTL_IPTAPE_GET_OUT_DATA

Function: Returns the values from the TTL output data registers.

Input: none

Output: IO_BITS structure

Notes: As with the previous call this can only be used with the 'B' or later PROM revision.



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
435 Park Dr.
Ben Lomond, CA 95005
831-336-8891
831-336-3840 fax
support@dyneng.com

All information provided is Copyright Dynamic Engineering

