

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

User Manual

IP-Generic Driver Documentation

Revision A

IP-Generic Generic IP Device Driver

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

©2003 by Dynamic Engineering.

Trademarks and registered trademarks are owned by their respective manufactures.
Manual Revision B. Revised October 21, 2003.

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	4
Note	4
Driver Installation	5
Driver Startup	6
IO Controls	9
IOCTL_IPGENERIC_GET_INFO	9
IOCTL_IPGENERIC_SET_MEM_OFFSET	9
IOCTL_IPGENERIC_GET_MEM_OFFSET	9
IOCTL_IPGENERIC_PUT_IO_DATA	10
IOCTL_IPGENERIC_GET_IO_DATA	10
IOCTL_IPGENERIC_SET_IP_CONTROL	10
IOCTL_IPGENERIC_GET_IP_CONTROL	10
IOCTL_IPGENERIC_REGISTER_EVENT	11
IOCTL_IPGENERIC_ENABLE_INTERRUPT	11
IOCTL_IPGENERIC_DISABLE_INTERRUPT	11
IOCTL_IPGENERIC_FORCE_INTERRUPT	11
IOCTL_IPGENERIC_GET_INT_STATUS	12
Write	12
Read	12
WARRANTY AND REPAIR	13
Service Policy	13
Out of Warranty Repairs	13
For Service Contact:	14



Introduction

The IpGeneric driver is a WindowsXP driver for a generic IP module. This driver can control any IP module by mapping the IO, MEM, and INT memory spaces so that they can be accessed by driver calls. A separate "Device Object" controls each IP module, and a separate handle references each Device Object. IO Control calls (IOCTLs) are used to configure the hardware and ReadFile() and WriteFile() calls are used to transfer data to and from the MEM space over the IP bus.

Note

This documentation will provide information about all calls made to the driver, and how the driver interacts with the device for each of these calls.



Driver Installation

There are several files provided in each driver drop. These files include IpGeneric.sys, IpDev.inf, DDIpGeneric.h, IpGenericGUID.h, IpGenericDef.h, IGTTest.exe, and IGTTest source files.

Copy IpDev.inf to the WINDOWS\INF folder and copy IpGeneric.sys to a floppy disk , or CD if preferred. Right click on the IpDev.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.

When the WindowsXP system sees the hardware for the first time it will start the *Found New Hardware Wizard* (if the IP carrier driver has not been installed, the system will be unable to see any IP devices). The **Unknown-IP Device** 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 *Unknown-IP 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 DDIpGeneric.h file is the C header file that defines the Application Program Interface (API) to the driver. The IpGenericGUID.h file is a C header file that defines the device interface identifier for the IpGeneric. These files are required at compile time by any application that wishes to interface with the IpGeneric driver. The IpGenericDef.h file contains the relevant bit defines for the IP module slot control register. These files are not needed for driver installation.

The IGTtest.exe file is a sample WindowsXP console application that makes calls into the lpGeneric 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 *IGTtest -dO -?* to display a list of commands (the IGTtest.exe file must be in the directory that the window is referencing). The commands are all of the form *IGTtest -dn -im* where *n* and *m* are the device number and driver ioctl number respectively.

Driver Startup

In WindowsXP once the driver has been installed it will start automatically when it sees 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 lpGenericGUID.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 hIpGeneric = 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_IPGENERIC,
                                  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_IPGENERIC,
                                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
hIpGeneric = CreateFile(deviceName,
                        GENERIC_READ | GENERIC_WRITE,
                        FILE_SHARE_READ | FILE_SHARE_WRITE,
                        NULL,
                        OPEN_EXISTING,
                        NULL,
                        NULL);

if(hIpGeneric == 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. 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_IPGENERIC_GET_INFO

Function: Returns the current driver version.

Input: none

Output: DRIVER_IP_DEVICE_INFO struct

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

IOCTL_IPGENERIC_SET_MEM_OFFSET

Function: Sets the address offset into the MEM space.

Input: ULONG

Output: none

Notes: Sets the address offset into the IP MEM space for ReadFile and WriteFile operations.

IOCTL_IPGENERIC_GET_MEM_OFFSET

Function: Returns the address offset into the MEM space.

Input: none

Output: ULONG

Notes: Returns the address offset into the IP MEM space for ReadFile and WriteFile operations.

IOCTL_IPGENERIC_PUT_IO_DATA

Function: Writes one word to the Rx FIFO.

Input: IO_ACCESS structure

Output: none

Notes: This call is used to write data to the IO space. The IO_ACCESS structure contains an address-offset field; a length field, which can be 1, 2, or 4; and a data field. For this call the address-offset, length, and data fields are initialized and the structure is passed to the driver which performs the write operation. See DDlpGeneric.h for the definition of the IO_ACCESS structure.

IOCTL_IPGENERIC_GET_IO_DATA

Function: Reads one word from the Tx FIFO.

Input: IO_ACCESS structure

Output: IO_ACCESS structure

Notes: This call is used to read data from the IO space. The IO_ACCESS structure contains an address-offset field; a length field, which can be 1, 2, or 4; and a data field. For this call the address-offset and length fields are initialized and the returned data field is written by the driver. See DDlpGeneric.h for the definition of the IO_ACCESS structure.

IOCTL_IPGENERIC_SET_IP_CONTROL

Function: Sets the configuration of the board slot.

Input: ULONG

Output: none

Notes: Controls the IP clock speed, access controls, interrupt enables, and the Force Interrupt bit for the IP slot that the board occupies. See the bit definitions in the IpGenericDef.h header file for more information.

IOCTL_IPGENERIC_GET_IP_CONTROL

Function: Returns the configuration of the board slot.

Input: none

Output: ULONG

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



IOCTL_IPGENERIC_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.

IOCTL_IPGENERIC_ENABLE_INTERRUPT

Function: Sets the master interrupt enable to true.

Input: INT_SEL enum type

Output: none

Notes: Sets one or both of the IP slot interrupt enables, leaving all other bit values in the IP slot control register the same. 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_IPGENERIC_DISABLE_INTERRUPT

Function: Sets the master interrupt enable to true.

Input: INT_SEL enum type

Output: none

Notes: Clears one or both of the IP slot interrupt enables, leaving all other bit values in the IP slot control register the same. This IOCTL is used when interrupt processing is no longer desired.

IOCTL_IPGENERIC_FORCE_INTERRUPT

Function: Causes a system interrupt to occur.

Input: none

Output: none

Notes: Causes an interrupt to be asserted for the IP slot. This IOCTL is used for development, to test interrupt processing.



IOCTL_IPGENERIC_GET_INT_STATUS

Function: Returns the interrupt status and interrupt vector.

Input: none

Output: INT_STAT struct

Notes: Returns the interrupt vector and the contents of the interrupt status register that were read in the last ISR call. These values are returned in the INT_STAT structure. See the bit definitions in the IpGenericDef.h header file and the struct definition in the DDIpGeneric.h header file for more information.

Write

Data to be written to the IP MEM space uses 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 is checked to see if the transfer length plus the address offset will overrun the end of memory. The driver takes advantage of the carrier's 32-bit double-write capability to load two IP words with a single PCI write until less than four bytes remain in the buffer. If necessary, the MEM address offset can be initialized using the IOCTL_IPGENERIC_SET_MEM_OFFSET call. See Win32 help files for details of the WriteFile() call.

Read

Data to be read from the IP MEM space uses a ReadFile() call. The user supplies the device handle, a pointer to the buffer to store the data in, 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 is checked to see if the transfer length plus the address offset will overrun the end of memory. The driver takes advantage of the carrier 32-bit double-read capability to read two IP words with a single PCI read until less than four bytes remain to be read. See Win32 help files for the details of the ReadFile() call.



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

