

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

PCI-NECL-ASN1

Driver Documentation

Win32 Driver Model

Revision A

Corresponding Hardware: Revision A/B

10-2004-0301/2

Corresponding Firmware: Revision A

PciECL
WDM Device Driver for the
PCI-NECL-ASN1
PCI based Re-configurable logic
with NECL and TTL IO

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

©2003-2004 by Dynamic Engineering.

Trademarks and registered trademarks are owned by their
respective manufacturers.
Manual Revision A. Revised August 23, 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 compatible user-provided equipment. Connection of incompatible hardware is likely to cause serious damage.



Table of Contents

Introduction	4
Note	4
Driver Installation	4
Windows 2000 Installation	4
Windows XP Installation	5
Driver Startup	6
IO Controls	8
IOCTL_PCIECL_GET_INFO	9
IOCTL_PCIECL_GET_STATUS	9
IOCTL_PCIECL_SET_CONFIG	9
IOCTL_PCIECL_GET_CONFIG	9
IOCTL_PCIECL_LOAD_PLL_DATA	10
IOCTL_PCIECL_READ_PLL_DATA	10
IOCTL_PCIECL_RESET_FIFOS	10
IOCTL_PCIECL_SET_EXT_FIFO_LEVELS	10
IOCTL_PCIECL_GET_EXT_FIFO_LEVELS	10
IOCTL_PCIECL_PUT_DATA	11
IOCTL_PCIECL_GET_DATA	11
IOCTL_PCIECL_REGISTER_EVENT	11
IOCTL_PCIECL_ENABLE_INTERRUPT	11
IOCTL_PCIECL_DISABLE_INTERRUPT	12
IOCTL_PCIECL_FORCE_INTERRUPT	12
Write	12
Read	12
WARRANTY AND REPAIR	13
Service Policy	13
Out of Warranty Repairs	13
For Service Contact:	14



Introduction

The PciECL driver is a Win32 driver model (WDM) device driver for the PCI-NECL-ASN1 from Dynamic Engineering. The PCI-NECL-ASN1 board has a PLX 9054 and a Xilinx FPGA to implement the PCI interface, DMA data I/O, 20 NECL, and 12 TTL data I/O for the board. There is also a programmable PLL that is programmed by and connected to the Xilinx to generate programmable clock rates for the I/O. An internal 1k x 32-bit FIFO is used to buffer the DMA transfers and a 128k x 32-bit external FIFO is used to buffer the ECL I/O.

When the PCI-NECL-ASN1 is recognized by the PCI bus configuration utility it will start the PciECL driver to allow communication with the device. IO Control calls (IOCTLs) are used to configure and read status from the PCI-NECL-ASN1. Read and Write calls are used to move blocks of data in and out of the device.

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. For more detailed information on the hardware implementation, refer to the PCI-NECL-ASN1 user manual (also referred to as the hardware manual).

Driver Installation

There are several files provided in each driver package. These files include PciECL.sys, PciECL.inf, DDPciECL.h, PciECLGUID.h, PciECLDef.h, PciECLTest.exe, and PciECLTest source files.

Windows 2000 Installation

Copy PciECL.inf and PciECL.sys to a floppy disk, or CD if preferred.

With the PCI-NECL-ASN1 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 PciECL.inf file.
- Select *Next*.
- Select *Finish* to close the *Found New Hardware Wizard*.

Windows XP Installation

Copy PciECL.inf to the WINDOWS\INF folder and copy PciECL.sys to a floppy disk, or CD if preferred. Right click on the PciECL.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 PCI-NECL-ASN1 installed, power-on the PCI host computer and wait for the *Found New Hardware Wizard* dialogue window to appear, or select the Add Hardware control panel.

- 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 *Pci-Generic-ECL 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*.

The DDPciECL.h file is a C header file that defines the Application Program Interface (API) to the driver. The PciECLGUID.h file is a C header file that defines the device interface identifier for the PciECL driver. These files are required at compile time by any application that wishes to interface with the PciECL driver. The PciECLDef.h file contains the relevant bit defines for the PciECL registers. These files are not needed for driver installation.

The PciECLTest.exe file is a sample Win32 console application that makes calls into the PciECL driver to test the driver calls without actually writing



any application code. It is not required during the driver installation. Open a command prompt console window and type *PciECLTest -dO -?* to display a list of commands (the PciECLTest.exe file must be in the directory that the window is referencing). The commands are all of the form *PciECLTest -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 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 PciECLGUID.h.

Below is example code for opening a handle for device O. The device number is underlined in the SetupDiEnumDeviceInterfaces call.

```
// The maximum length of the device name for a given interface
#define MAX_DEVICE_NAME 256
// Handle to the device object
HANDLE hPciECL = 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_PCIECL,
                                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_PCIECL,
                                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
lstrcpy(deviceName,
        pDeviceDetail->DevicePath,
        MAX_DEVICE_NAME);

// Cleanup search
free(pDeviceDetail);
SetupDiDestroyDeviceInfoList(hDeviceInfo);

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

if(hPciECL == 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 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. The IOCTLs defined in this driver are as follows:



IOCTL_PCIECL_GET_INFO

Function: Return the Driver Version, Switch value, Instance Number, and External FIFO size.

Input: none

Output: PCIECL_DRIVER_DEVICE_INFO structure

Notes: Switch value is the configuration of the onboard dipswitch that has been selected by the User (see the board silk screen for bit position and polarity). The FIFO size is dynamically detected when the driver starts up. The value returned is one less than the actual FIFO size (the index of the last word).

IOCTL_PCIECL_GET_STATUS

Function: Return the DMA FIFO level and other status information.

Input: none

Output: Unsigned long int

Notes: The value read from the status register contains the DMA FIFO word count and other status flag values for both FIFOs (see the bit definitions in PciECLDef.h for information on interpreting this value).

IOCTL_PCIECL_SET_CONFIG

Function: Write to the base configuration register on the PCI-NECL-ASN1.

Input: Unsigned long int

Output: none

Notes: Only the bits in the BASE_CONFIG_MASK are controlled by this command. See the bit definitions in PciECLDef.h for information on determining this value.

IOCTL_PCIECL_GET_CONFIG

Function: Return the configuration of the base control register.

Input: none

Output: Unsigned long int

Notes: The value read does not include reset bits or the force interrupt bit. This command is used mainly for testing.



IOCTL_PCIECL_LOAD_PLL_DATA

Function: Load the internal registers of the PLL.

Input: PCIECL_PLL_DATA structure

Output: none

Notes: The PCIECL_PLL_DATA structure has one field: An array of 40 bytes containing the PLL register data to write.

IOCTL_PCIECL_READ_PLL_DATA

Function: Return the contents of the PLL's internal registers.

Input: none

Output: PCIECL_PLL_DATA structure

Notes: The register data is output in the PCIECL_PLL_DATA structure as an array of 40 bytes.

IOCTL_PCIECL_RESET_FIFOS

Function: Reset the External and/or DMA FIFO.

Input: PCIECL_FIFO_SEL enumeration type

Output: none

Notes: Resets either the DMA FIFO, the External FIFO, or both depending on the input value.

IOCTL_PCIECL_SET_EXT_FIFO_LEVELS

Function: Set the External FIFO almost empty and almost full levels.

Input: PCIECL_EXT_LEVEL_LOAD structure

Output: none

Notes: The PCIECL_EXT_LEVEL_LOAD structure has two fields: AlmostFull – the almost full level to set in the External FIFO, and AlmostEmpty – the almost empty level to set in the External FIFO.

IOCTL_PCIECL_GET_EXT_FIFO_LEVELS

Function: Return the External FIFO almost empty and almost full levels.

Input: none

Output: PCIECL_EXT_LEVEL_LOAD structure

Notes: See above for description of PCIECL_EXT_LEVEL_LOAD structure.



IOCTL_PCIECL_PUT_DATA

Function: Write one long word to the PciECL memory space.

Input: PCIECL_DATA_LOAD structure

Output: none

Notes: The PCIECL_DATA_LOAD structure has two long word fields: Address – the address offset to write to, and Data – the data value to write.

IOCTL_PCIECL_GET_DATA

Function: Read one long word from the PciECL memory space.

Input: Unsigned long int

Output: Unsigned long int

Notes: The address offset to read from is passed to this command and a long word of data read from the specified address offset is returned.

IOCTL_PCIECL_REGISTER_EVENT

Function: Register 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 a user interrupt is serviced. The user interrupt service routine waits on this event, allowing it to respond to the interrupt. The DMA interrupts do not cause the event to be signaled.

IOCTL_PCIECL_ENABLE_INTERRUPT

Function: Enable the master interrupt.

Input: none

Output: none

Notes: This command must be run to allow the board to respond to local interrupts. The master interrupt enable is disabled in the driver interrupt service routine. This command must be run to re-enable it.



IOCTL_PCIECL_DISABLE_INTERRUPT

Function: Disable the master interrupt.

Input: none

Output: none

Notes: Used when local interrupt processing is no longer desired.

IOCTL_PCIECL_FORCE_INTERRUPT

Function: Causes a system interrupt to occur.

Input: none

Output: none

Notes: Causes an interrupt to be asserted on the PCI bus as long as the master interrupt is enabled. This IOCTL is used for development, to test interrupt processing.

Write

PCI-NECL-ASN1 DMA data is written to the device using the write command. Writes are executed using the Win32 function WriteFile() and passing in the handle to the device opened with CreateFile(), a pointer to a pre-allocated buffer containing the data to be written, an unsigned long int that represents the size of that buffer in bytes, a pointer to an unsigned long int to contain the number of bytes actually written, and a pointer to an optional Overlapped structure for performing asynchronous IO.

Read

PCI-NECL-ASN1 DMA data is read from the device using the read command. Reads are executed using the Win32 function ReadFile() and passing in the handle to the device opened with CreateFile(), a pointer to a pre-allocated buffer that will contain the data read, an unsigned long int that represents the size of that buffer in bytes, a pointer to an unsigned long int to contain the number of bytes actually read, and a pointer to an optional Overlapped structure for performing asynchronous IO.



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

