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-XG1

Driver Documentation

Win32 Driver Model

Revision A
Corresponding Hardware: Revision A/B
10-2004-0301/2
Corresponding Firmware: Revision A

PciSE
WDM Device Driver for the
PCI-NECL-XG1
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

Manual Revision A. Revised August 23, 2004.

Table of Contents

Introduction

The PciSE driver is a Win32 driver model (WDM) device driver for the PCI-NECL-XG1 from Dynamic Engineering. The PCI-NECL-XG1 board has a PLX 9054 and a Xilinx FPGA to implement the PCI interface, DMA data I/O, 19 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 high-speed serial I/O.

When the PCI-NECL-XG1 is recognized by the PCI bus configuration utility it will start the PciSE driver to allow communication with the device. IO Control calls (IOCTLs) are used to configure and read status from the PCI-NECL-XG1. 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-XG1 user manual (also referred to as the hardware manual).

Driver Installation

There are several files provided in each driver package. These files include PciSE.sys, PciSE.inf, DDPciSE.h, PciSEGUID.h, PciSEDef.h, PSETest.exe, and PSETest source files.

Windows 2000 Installation

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

With the PCI-NECL-XG1 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 PciSE.inf file.
• Select *Next*.
• Select *Finish* to close the *Found New Hardware Wizard*.

Windows XP Installation

Copy PciSE.inf to the WINDOWS\INF folder and copy PciSE.sys to a floppy disk, or CD if preferred. Right click on the PciSE.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-XG1 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-Serial-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 DDPciSE.h file is a C header file that defines the Application Program Interface (API) to the driver. The PciSEGUID.h file is a C header file that defines the device interface identifier for the PciSE driver. These files are required at compile time by any application that wishes to interface with the PciSE driver. The PciSEDef.h file contains the relevant bit defines for the PciSE registers. These files are not needed for driver installation.

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

Below is example code for opening a handle for device 0. 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                              hPciSE = 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_PCISE,
                                    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_PCISE,
                                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
hPciSE = CreateFile(deviceName,
                    GENERIC_READ    | GENERIC_WRITE,
                    FILE_SHARE_READ | FILE_SHARE_WRITE,
                    NULL,
                    OPEN_EXISTING,
                    NULL,
                    NULL);

if(hPciSE == 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_PCISE_GET_INFO

Function: Return the Driver Version, Switch value, Instance Number, and External FIFO size.
Input: none
Output: PCISE_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_PCISE_GET_STATUS

Function: Return the FIFO levels and other status information.
Input: none
Output: PCISE_READ_STATUS structure
Notes: The PCISE_READ_STATUS structure has two fields: StatusReg is the value read from the status register, which contains the DMA FIFO word count and other status flag values for both FIFOs (see the bit definitions in PciSEDef.h for information on interpreting this value) and ExtFifoCount which is the number of words in the External FIFO.

IOCTL_PCISE_SET_CONFIG

Function: Write to the base configuration register on the PCI-Serial-ECL.
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 PciSEDef.h for information on determining this value.

IOCTL_PCISE_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_PCISE_SET_INT_CONFIG

Function: Set the Interrupt enable configuration.
Input: Unsigned long int
Output: none
Notes: This command determines which conditions are enabled to cause an interrupt when the master interrupt enable is set. See the bit definitions in PciSEDef.h for information on determining this value.

IOCTL_PCISE_GET_INT_CONFIG

Function: Return the Interrupt enable configuration.
Input: none
Output: Unsigned long int
Notes: Returns the signals enabled to cause an interrupt. See the bit definitions in PciSEDef.h for information on interpreting this value.

IOCTL_PCISE_GET_INT_STAT

Function: Return the interrupt status and clear the latched bits.
Input: none
Output: Unsigned long int
Notes: This command returns the latched interrupt status bits and the interrupt active status bit. Latched bits that are read as true are then automatically written back to the register to clear the latches. This prevents missing interrupts that occur between the read and the write of the register. See the bit definitions in PciSEDef.h for information on interpreting this value.

IOCTL_PCISE_PUT_TX_DATA

Function: Load a Tx data word.
Input: Unsigned long int
Output: none
Notes: This command can be used to load a single long word to the External FIFO.

IOCTL_PCISE_GET_RX_DATA

Function: Read an Rx data word.
Input: none
Output: Unsigned long int
Notes: This command can be used to read a single long word from the External FIFO.

IOCTL_PCISE_PUT_DMA_DATA

Function: Load a DMA FIFO data word.
Input: Unsigned long int
Output: none
Notes: Loads a single long word into the DMA FIFO.

IOCTL_PCISE_GET_DMA_DATA

Function: Read a DMA FIFO data word.
Input: none
Output: Unsigned long int
Notes: Reads a single long word from the DMA FIFO.

IOCTL_PCISE_RESET_FIFOS

Function: Reset the External and/or DMA FIFO.
Input: PCISE_FIFO_SEL enumeration type
Output: none
Notes: Resets either the DMA FIFO, the External FIFO, or both depending on the input value.

IOCTL_PCISE_SET_EXT_FIFO_LEVELS

Function: Set the External FIFO almost empty and almost full levels.
Input: PCISE_EXT_LEVEL_LOAD structure
Output: none
Notes: The PCISE_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_PCISE_GET_EXT_FIFO_LEVELS

Function: Return the External FIFO almost empty and almost full levels.
Input: none
Output: PCISE_EXT_LEVEL_LOAD structure
Notes: See above for description of PCISE_EXT_LEVEL_LOAD structure.

IOCTL_PCISE_SET_DMA_FIFO_LEVELS

Function: Set the DMA FIFO almost empty and almost full levels.
Input: PCISE_DMA_LEVEL_LOAD structure
Output: none
Notes: The PCISE_DMA_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_PCISE_GET_DMA_FIFO_LEVELS

Function: Return the DMA FIFO almost empty and almost full levels.
Input: none
Output: PCISE_DMA_LEVEL_LOAD structure
Notes: See above for description of PCISE_DMA_LEVEL_LOAD structure.

IOCTL_PCISE_GET_ISR_STATUS

Function: Return the interrupt status read in the ISR from the last interrupt.
Input: none
Output: Unsigned long int
Notes: The value returned is the result of the last interrupt caused by one of the signals enabled in the previous IOCTL_PCISE_SET_INT_CONFIG command. The interrupts that deal with the DMA transfers do not affect this value.

IOCTL_PCISE_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_PCISE_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_PCISE_DISABLE_INTERRUPT

Function: Disable the master interrupt.
Input: none
Output: none
Notes: Used when local interrupt processing is no longer desired.

IOCTL_PCISE_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.

IOCTL_PCISE_LOAD_PLL_DATA

Function: Load the internal registers of the PLL.
Input: PCISE_PLL_DATA structure
Output: none
Notes: The PCISE_PLL_DATA structure has one field: an array of 40 bytes containing the PLL register data to write.

IOCTL_PCISE_READ_PLL_DATA

Function: Return the contents of the PLL's internal registers.
Input: none
Output: PCISE_PLL_DATA structure
Notes: The register data is output in the PCISE_PLL_DATA structure as an array or 40 bytes.

IOCTL_PCISE_SET_TTL

Function: Sets the values of the 12 TTL lines.
Input: Unsigned short int
Output: none
Notes: These are open drain lines that are pulled-up to +5 volts, therefore they must be set high in order to be used as inputs.

IOCTL_PCISE_GET_TTL

Function: Returns the values read from the 12 TTL lines.
Input: none
Output: Unsigned short int
Notes: These are open drain lines that are pulled-up to +5 volts, therefore they must be set high in order to be used as inputs, otherwise a low will be read regardless of the input level.

IOCTL_PCISE_SET_ECL

Function: Set the values of the lower 16 ECL output lines.
Input: Unsigned short int
Output: none
Notes: The lower 16 bits of the ECL I/O are used as a general-purpose data bus. This call allows setting the output value of this bus.

IOCTL_PCISE_GET_ECL

Function: Return the values read from the lower 16 ECL input lines.
Input: none
Output: Unsigned short int
Notes: The lower 16 bits of the ECL I/O are used as a general-purpose data bus. This call allows reading the value of the input bus.

IOCTL_PCISE_SET_TX_COUNT

Function: Set the start value of the Tx data counter.
Input: none
Output: Unsigned long int
Notes: In order to test the card in a standalone configuration, The transmitter can be configured to output data from a counter rather than the External FIFO. This allows the External FIFO to be used by the receiver so that when the transmitter is looped to the receiver, both can operate simultaneously. This call sets the start value of the transmitter data counter.

Write

PCI-NECL-XG1 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-XG1 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