DYNAMIC ENGINEERING
150 DuBois St., Suite C, Santa Cruz, CA 95060
831-457-8891 **Fax** 831-457-4793
http://www.dyneng.com
sales@dyneng.com
Est. 1988

PCI-NECL-STE1

Driver Documentation

Win32 Driver Model

Revision B
Corresponding Hardware: Revision B
10-2004-0302
Corresponding Firmware: Revision C

PciNeclSte1
WDM Device Driver for the
PCI-NECL-STE1
PCI based Bidirectional DMA
With NECL and TTL I/O

Dynamic Engineering
150 DuBois St., Suite C
Santa Cruz, CA 95060
831-457-8891
831-457-4793 FAX

Manual Revision B. Revised June 25, 2010.

Table of Contents

Introduction

The PciNeclSte1 driver is a Win32 driver model (WDM) device driver for the PCI-NECL-STE1 from Dynamic Engineering. The PCI-NECL-STE1 board has a PLX PCI 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. The board has an internal 2k x 32-bit FIFO and an external 128k x 32-bit FIFO. Either one of these FIFOs can be configured by software to be used for transmitter data transfers. The remaining FIFO will be used for receiver data transfers.

When the PCI-NECL-STE1 is recognized by the PCI bus configuration utility it will start the PciNeclSte1 driver to allow communication with the device. IO Control calls (IOCTLs) are used to configure and read status from the PCI-NECL-STE1. 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-STE1 user manual (also referred to as the hardware manual).

Driver Installation

There are several files provided in each driver package. These files include PciNeclSte1.sys, PciNeclSte1.inf, DDPciNeclSte1.h, PciNeclSte1GUID.h, PlxDef.h, PciNeclSte1Test.exe, and PciNeclSte1Test source files.

Windows 2000 Installation

Copy PciNeclSte1.inf and PciNeclSte1.sys to a floppy disk, CD or some other accessible location.

With the PCI-NECL-STE1 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 PciNeclSte1.inf file.
- Select ***Next***.
- Select ***Finish*** to close the ***Found New Hardware Wizard***.

Windows XP Installation

Copy PciNeclSte1.inf and PciNeclSte1.sys to a floppy disk, CD or some other accessible location.

With the PCI-NECL-STE1 hardware installed, power-on the PCI host computer and wait for the ***Found New Hardware Wizard*** dialogue window to appear.

- Insert the disk prepared above in the desired drive.
- Select ***No when asked to connect to Windows Update***.
- Select ***Next***.
- Select ***Install the software automatically.***
- Select ***Next***.
- Select ***Finish*** to close the ***Found New Hardware Wizard***.

DDPciNeclSte1.h is a C header file that defines the Application Program Interface (API) to the driver. PciNeclSte1GUID.h is a C header file that defines the device interface identifier for the PciNeclSte1 driver. PlxDef.h contains the relevant address offsets and bit defines for the PLX PCI 9054 internal registers. These files are required at compile time by any application that wishes to interface with the PciNeclSte1 driver, but are not needed for driver installation.

PciNeclSte1Test.exe is a sample Win32 console application that makes calls into the PciNeclSte1 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 ***PciNeclSte1Test –d0 -?*** to display a list of commands (the PciNeclSte1Test.exe file must be in the directory that the window is referencing). The commands are all of the form ***PciNeclSte1Test –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 individual 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 PciNeclSte1GUID.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

// Device handle
HANDLE                           hPciNeclSte1 = INVALID_HANDLE_VALUE;
// Enumeration index
ULONG                            i;
// Number of PciNeclSte1 devices installed
UCHAR                            numDevs;
// PciNeclSte1 device number
ULONG                            devNum;
// 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 reguired 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;
BOOLEAN                          found = TRUE;

hDeviceInfo =
   SetupDiGetClassDevs((LPGUID)&GUID_DEVINTERFACE_PCI_NECL_STE1,
                               NULL,
                               NULL,
                               DIGCF_PRESENT | DIGCF_DEVICEINTERFACE);

if(hDeviceInfo == INVALID_HANDLE_VALUE)
{
   status = GetLastError();
   printf("**Error: couldn't get class info, (%d)\n", status); exit(-1);
}

i = 0;
interfaceData.cbSize = sizeof(interfaceData);

for(i = 0; i <= devNum; i++)
{// Find the interface for device devNum
   if(!SetupDiEnumDeviceInterfaces(hDeviceInfo,
                                   NULL,
                           (LPGUID)&GUID_DEVINTERFACE_PCI_NECL_STE1,
                                   i,
                                   &interfaceData))
```

```
    {
        status = GetLastError();
        if(status == ERROR_NO_MORE_ITEMS)
        {
            printf("**Error: couldn't find device(no more items), (%d)\n", i);
            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
hPciNeclSte1 = CreateFile(deviceName,
                          GENERIC_READ    | GENERIC_WRITE,
                          FILE_SHARE_READ | FILE_SHARE_WRITE,
                          NULL,
                          OPEN_EXISTING,
                          NULL,
                          NULL);

if(hPciNeclSte1 == 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.

```
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
```

The IOCTLs defined in this driver are as follows:

IOCTL_PCI_NECL_STE1_GET_INFO

Function: Returns the Driver and Xilinx Version, Switch value, Instance Number, External FIFO size and PLL device ID.
Input: None
Output: PCI_NECL_STE1_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 instance number is the zero-based device number in the order evaluated by the driver. The PLL ID is set by the manufacture and is either 0x69 or 0x6A.

IOCTL_PCI_NECL_STE1_SET_CONFIG

Function: Writes a value to the base control register on the PCI-NECL-STE1.
Input: Unsigned long integer
Output: None
Notes: Only the bits in the BASE_CONFIG_MASK are controlled by this command. See the bit definitions in DDPciNeclSte1.h for information on determining this value.

IOCTL_PCI_NECL_STE1_GET_CONFIG

Function: Returns the configuration of the base control register.
Input: None
Output: Unsigned long integer
Notes: Only the bits in the BASE_READ_MASK are included in the return value. This includes the configuration bits set in the previous call, plus the master interrupt enable bit. See the bit definitions in DDPciNeclSte1.h for information on interpreting this value.

IOCTL_PCI_NECL_STE1_GET_STATUS

Function: Returns status information for the FIFOs and interrupt latches.
Input: None
Output: Unsigned long integer
Notes: The value read from the status register contains status flag values for both FIFOs and the state of the interrupt condition latches even if the interrupt condition is not enabled. The latched status bits will be automatically cleared by this call. Latches will only be cleared if they were set when the status was read. This prevents missing a latched status condition that becomes valid in the time between the read and write to this register. See the bit definitions in DDPciNeclSte1.h for information on interpreting this value.

IOCTL_PCI_NECL_STE1_LOAD_PLL_DATA

Function: Loads the internal registers of the PLL device.
Input: PCI_NECL_STE1_PLL_DATA structure
Output: None
Notes: The PCI_NECL_STE1_PLL_DATA structure has one field: An array of 40 bytes containing the PLL register data to write. The UserTestApp includes a function that reads the .jed file generated by the CyberClock utility from Cypress Semiconductor and returns the data array required by this call.

IOCTL_PCI_NECL_STE1_READ_PLL_DATA

Function: Returns the contents of the PLL device's internal registers.
Input: None
Output: PCI_NECL_STE1_PLL_DATA structure
Notes: The register data is output in the PCI_NECL_STE1_PLL_DATA structure as an array or 40 bytes.

IOCTL_PCI_NECL_STE1_RESET_FIFOS

Function: Resets the transmit and/or receive FIFO.
Input: PCI_NECL_STE1_FIFO_SEL enumeration type
Output: None
Notes: Resets either the transmit FIFO, the receive FIFO, or both depending on the input value.

IOCTL_PCI_NECL_STE1_SET_FIFO_LEVELS

Function: Sets the transmit FIFO almost empty and receive FIFO almost full levels.
Input: PCI_NECL_STE1_FIFO_LEVELS structure
Output: None
Notes: The PCI_NECL_STE1_FIFO_LEVELS structure has two fields: AlmostFull – the almost full level to set in the receive (external) FIFO, and AlmostEmpty – the almost empty level to set in the transmit (internal) FIFO. The values are both absolute word counts above zero (empty).

IOCTL_PCI_NECL_STE1_GET_FIFO_LEVELS

Function: Returns the transmit FIFO almost empty and receive FIFO almost full levels.
Input: None
Output: PCI_NECL_STE1_FIFO_LEVELS structure
Notes: See above for description of PCI_NECL_STE1_FIFO_LEVELS structure.

IOCTL_PCI_NECL_STE1_GET_FIFO_COUNTS

Function: Returns the current word-counts of the transmit and receive FIFOs.
Input: None
Output: PCI_NECL_STE1_FIFO_COUNTS structure
Notes: See DDPciNeclSte1.h for the description of PCI_NECL_STE1_FIFO_COUNTS.

IOCTL_PCI_NECL_STE1_WRITE_FIFO

Function: Writes one long-word to the transmit FIFO.
Input: Unsigned long integer
Output: None
Notes: Used to make single-word accesses to the transmit FIFO instead of using DMA.

IOCTL_PCI_NECL_STE1_READ_FIFO

Function: Reads one long-word from the receive FIFO.
Input: None
Output: Unsigned long integer
Notes: Used to make single-word accesses from the receive FIFO instead of using DMA.

IOCTL_PCI_NECL_STE1_START_TRANSMIT

Function: Enables the transmitter to start sending data.
Input: None
Output: None
Notes: If the transmit FIFO is empty the transmit state-machine will wait for data to be available before starting the transmission. Once the transmission begins, when the transmit FIFO data is exhausted, the done status bit will be set and, if enabled, the transmit start bit will be automatically cleared. To prevent the transmitter stopping too soon, it would be prudent to load some amount of data into the transmit FIFO prior to making this call.

IOCTL_PCI_NECL_STE1_START_RECEIVE

Function: Enables the receiver to start receiving data.
Input: None
Output: None
Notes: The reception is qualified with the data enable signal. A byte is captured for each clock while the enable is high. Data is assembled into long-words, least significant byte first, and stored in the receive FIFO. If the enable goes low while before a long-word is complete, the word is written to the FIFO with zeros in the upper byte(s).

IOCTL_PCI_NECL_STE1_STOP_TRANSMIT

Function: Stops the transmission.
Input: None
Output: None
Notes: Clears the transmit start bit. This call is not needed if the BASE_TX_CLEAR_EN bit is set unless the transmission is to be prematurely terminated.

IOCTL_PCI_NECL_STE1_STOP_RECEIVE

Function: Stops the reception.
Input: None
Output: None
Notes: This call is used when data reception is no longer desired.

IOCTL_PCI_NECL_STE1_SET_TTL

Function: Sets the value of the TTL control register to be driven onto the 12 TTL lines.
Input: Unsigned long integer
Output: None
Notes: The lowest 12 bits of this register are applied to the inputs of the TTL drivers. The outputs of these drivers are “wire-ored” with any other driver on the external TTL bus. If a data-bit is set high, the driver is tri-stated and the bus line is pulled high by a resistor to +5 volts. In this case another driver on the bus is able to determine the state of the bus-line. If a data-bit is set low, the bus line is driven low regardless of the state of other drivers on the bus.

IOCTL_PCI_NECL_STE1_GET_TTL

Function: Returns the value of the TTL control register set in the previous call.
Input: None
Output: Unsigned long integer
Notes: The return value of this call does not necessarily reflect the state of the external TTL lines, as remote drivers can drive the bus low when a local control bit is set high.

IOCTL_PCI_NECL_STE1_READ_TTL

Function: Reads and returns the state of the external TTL lines.
Input: None
Output: Unsigned long integer
Notes: The value returned by this call reflects the actual state of the external TTL lines. Depending on the activity of remote drivers, it may be different from the value returned by the previous call.

IOCTL_PCI_NECL_STE1_SET_ECL

Function: Sets the value of the ECL control register to be driven onto the eight auxiliary ECL lines.
Input: Unsigned long integer
Output: None
Notes: ECL output data lines 11-17 and 19 are unused in this design. They are collected into an eight-bit bus that is controlled by bits 0-7 of the ECL control register. Unlike the TTL bus above, these bits are not bi-directional; so, if used, they must be connected point-to-point.

IOCTL_PCI_NECL_STE1_GET_ECL

Function: Returns the value of the ECL control register set in the previous call.
Input: None
Output: Unsigned long integer
Notes: The return value of this call should match the value written in the previous call. It does not report the state of the external ECL lines.

IOCTL_PCI_NECL_STE1_READ_ECL

Function: Reads and returns the state of the eight auxiliary external ECL lines.
Input: None
Output: Unsigned long integer
Notes: The value returned by this call reflects the state of the eight auxiliary ECL lines.

IOCTL_PCI_NECL_STE1_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. This mechanism is only for interrupts local to the PCI 9054⇔Xilinx bus, such as FIFO levels and transmit/receive state-machine events. The DMA interrupts are handled internally by the PLX PCI 9054.

IOCTL_PCI_NECL_STE1_ENABLE_INTERRUPT

Function: Enables 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 after each user interrupt is serviced.

IOCTL_PCI_NECL_STE1_DISABLE_INTERRUPT

Function: Disables the master interrupt.
Input: None
Output: None
Notes: This call is used when local interrupt processing is no longer desired.

IOCTL_PCI_NECL_STE1_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 and the PLX PCI 9054 is configured to respond to local interrupts. This IOCTL is used for development, to test interrupt processing.

IOCTL_PCI_NECL_STE1_GET_ISR_STATUS

Function: Returns the value of the status register that was read in the driver interrupt service routine when the last user interrupt was serviced.
Input: None
Output: Unsigned long integer
Notes: The latched status bits will have been cleared automatically in the driver interrupt service routine. So this call is useful, if multiple interrupt conditions are enabled, to determine which condition caused the interrupt. See the bit definitions in DDPciNeclSte1.h for information on interpreting this value.

Write

PCI-NECL-STE1 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 integer that represents the number of bytes to be written, a pointer to an unsigned long integer to contain the number of bytes actually written, and a pointer to an optional Overlapped structure for performing asynchronous IO.

Read

PCI--NECL-STE1 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 integer that represents the number of bytes to be read, a pointer to an unsigned long integer to contain the number of bytes actually read, and a pointer to an optional Overlapped structure for performing asynchronous IO. The read DMA (channel 0) is configured for demand-mode DMA, which means that it will not request the PCI bus until there is actually data in the receive FIFO to be transferred. Also when the receive FIFO becomes close to empty (below 4 words), the DMA will pause for several micro-seconds to allow sufficient data to accumulate for a reasonable burst before requesting the bus again. This promotes more efficient use of the PCI bus bandwidth.

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 merchantability 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
150 DuBois St., Suite C
Santa Cruz, CA 95060
831-457-8891
831-457-4793 Fax
support@dyneng.com