DYNAMIC ENGINEERING
150 DuBois, Suite 3 Santa Cruz, CA 95060
(831) 457-8891 **Fax** (831) 457-4793
www.dyneng.com
sales@dyneng.com
Est. 1988

PciAlt

Driver Documentation

Win32 Driver Model

Revision D
Corresponding Hardware: Revision F-H
10-2002-0706/0707/0708
Corresponding Firmware: Revision J-L

PciAlt

WDM Device Driver for the
PCI-Altera-485/LVDS
PCI based Re-configurable logic
with RS-485/LVDS and TTL IO

Dynamic Engineering
150 DuBois, Suite 3
Santa Cruz, CA 95060
(831) 457-8891
FAX: (831) 457-4793

Manual Revision D. Revised December 3, 2007.

Table of Contents

Introduction

The PciAlt driver is a Win32 driver model (WDM) device driver for the PCI-Altera-485/LVDS from Dynamic Engineering. The PCI-Altera board has a PLX 9054 and a Xilinx FPGA to implement the PCI interface and DMA data I/O for the board. In addition there is an Altera EP20K400EBC652 FPGA that is programmed from the PCI interface with a configuration file that is resident on the host hard drive. The Altera controls 40 RS-485 or LVDS transceivers and 12 TTL I/O lines. There are also 8 programmable PLLs with three clock outputs each that are programmed through the Altera and connected to 24 clock input pins on the Altera FPGA. Eight transmit FIFOs and eight receive FIFOs buffer data between the Xilinx and the Altera for eight independent I/O channels.

When the PCI-Altera is recognized by the PCI bus configuration utility it will start the PciAlt driver, which reads the registry to locate the default Altera configuration file and program the Altera FPGA. The location and name of this configuration file is defined in the PciAlt.inf file. The default location and file name is: \SystemRoot\AlteraDesigns\AltATP.rbf where SystemRoot is the windows directory on the host machine. The Altera design ID value is read from the Altera base address and the appropriate Altera driver is loaded to control the various Altera functions. In order to properly load the Altera on power-up, create an AlteraDesigns directory under the Windows directory and place the AltATP.rbf file in this directory. Any other Altera design files to be loaded later should also be placed in this directory.

The Altera is treated as a hot-swappable child of the PciAlt parent. This means that different Altera configurations can be loaded at any time without powering down. The new Altera driver will then be loaded automatically. A separate handle to the PciAlt and to the current Altera driver can be obtained using CreateFile() calls (see below). IO Control calls (IOCTLs) are used to configure the PCI-Altera and read status and Read and Write calls are used to move blocks of data in and out. The Control Calls specific to the current Altera design are described in the appropriate Altera driver manual.

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-Altera user manual (also referred to as the hardware manual).

Driver Installation

There are several files provided in each driver package. These files include PciAlt.sys, PciAlt.inf, DDPciAlt.h, PciAltGUID.h, PlxDef.h, PATest.exe, and PATest source files.

DDPciAlt.h is a C header file that defines the Application Program Interface (API) to the driver. PciAltGUID.h is a C header file that defines the device interface identifier for the PciAlt driver. These files are required at compile time by any application that wishes to interface with the PciAlt driver, but are not required for driver installation. The PlxDef.h file contains the relevant bit defines and offsets for the PLX registers. This file is also not needed for driver installation.

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

Windows 2000 Installation

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

With the PCI-Altera board 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 PciAlt.inf file.
- Select ***Next***.
- Select ***Finish*** to close the ***Found New Hardware Wizard***.

Windows XP Installation

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

With the PCI-Altera board installed, power-on the PCI host computer and wait for the 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***.

At this point, the wizard will reactivate to install the Altera design driver. This process is similar, and details can be found in the appropriate Altera driver manual.

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 PciAltGUID.h.

Below is example code for opening a handle for device *devNum*.

```
// The maximum length of the device name for a given interface
#define MAX_DEVICE_NAME 256

// Device handle
HANDLE                           hPciAlt = INVALID_HANDLE_VALUE;

// Return status from command
LONG                             status;

// Base configuration value
ULONG                            config;

// Handle to device information structure for the interface to devices
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;

hDeviceInfo = SetupDiGetClassDevs(
                          (LPGUID)&GUID_DEVINTERFACE_PCI_ALT,
                                  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);
}

interfaceData.cbSize = sizeof(interfaceData);

// Find the interface for device devNum
if(!SetupDiEnumDeviceInterfaces(hDeviceInfo,
                                NULL,
                        (LPGUID)&GUID_DEVINTERFACE_PCI_ALT,
                                devNum,
                                &interfaceData))
{
   status = GetLastError();
   if(status == ERROR_NO_MORE_ITEMS)
   {
      printf("**Error: couldn't find device(no more items), (%d)\n", devNum);
      SetupDiDestroyDeviceInfoList(hDeviceInfo);
      exit(-1);
   }
   else
   {
      printf("**Error: couldn't enum device, (%d)\n", status);
      SetupDiDestroyDeviceInfoList(hDeviceInfo);
      exit(-1);
   }
}

// Found our device, 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
hPciAlt = CreateFile(deviceName,
                     GENERIC_READ    | GENERIC_WRITE,
                     FILE_SHARE_READ | FILE_SHARE_WRITE,
                     NULL,
                     OPEN_EXISTING,
                     NULL,
                     NULL);

if(hPciAlt == 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_PCI_ALT_GET_INFO

Function: Return the Driver Version, Switch value, Instance Number, and Transmit and Receive FIFO sizes.
Input: None
Output: 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 sizes are dynamically detected when the driver starts up. The value returned is one less than the actual FIFO size (the index of the last byte).

IOCTL_PCI_ALT_GET_STATUS

Function: Return the FIFO levels and other status information.
Input: None
Output: Unsigned long integer
Notes: See the bit definitions in DDPciAlt.h for information on interpreting this value.

IOCTL_PCI_ALT_SET_CONFIG

Function: Writes to the base configuration register on the PCI-Altera.
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 DDPciAlt.h for information on determining this value.

IOCTL_PCI_ALT_GET_CONFIG

Function: Returns the configuration of the base control register.
Input: None
Output: Unsigned long integer
Notes: The value read does not include reset bits, the Altera load bit, or the force interrupt bit. This command is used mainly for testing.

IOCTL_PCI_ALT_LOAD_COMMAND

Function: Load a write_Tx / read_Rx command.
Input: TRANSFER_COMMAND structure
Output: None
Notes: This command is used to specify Tx/Rx data routing and must be coordinated with a compatible PCI data transfer. The fields of the TRANSFER_COMMAND structure are: Channels – an eight-bit mask of the Tx/Rx channels. For a receive command only one bit may be set, as only one FIFO can be read at a time, however for a transmit command any number of bits may be set (broadcast mode). Count – the number of long words to transfer (should not exceed the target FIFO size divided by four). And Transmit – true for a transmit command, false for a receive command.

IOCTL_PCI_ALT_GET_COMMAND_STAT

Function: Return the command status and count.
Input: None
Output: COMMAND_STAT structure
Notes: The COMMAND_STAT structure has two fields: Count is the number of commands currently in the command queue and Status is a combination of six values – the number of words in the DMA FIFO, an eight-bit mask of the active channel(s), whether a command is currently active, the direction of the transfer if a command is active, whether the command queue is full, and whether a command is ready to execute. See the bit definitions in DDPciAlt.h for information on interpreting this value.

IOCTL_PCI_ALT_RESET_COMMAND_QUEUE

Function: Reset the command queue.
Input: None
Output: None
Notes: This command empties the command queue. All commands that were pending execution will be lost and the command count will be set to zero.

IOCTL_PCI_ALT_SET_INT_CONFIG

Function: Set the Interrupt enable configuration.
Input: Unsigned long integer
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 DDPciAlt.h for information on determining this value.

IOCTL_PCI_ALT_GET_INT_CONFIG

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

IOCTL_PCI_ALT_GET_INT_STAT

Function: Return the interrupt status and clear the latched bits.
Input: None
Output: Unsigned long integer
Notes: This command returns the latched interrupt status bits and the interrupt active status bit. Latched bits that are read as true are then cleared by writing only those bits back to the interrupt status register. This prevents missing interrupts that occur between the read and the write of the register. See the bit definitions in DDPciAlt.h for information on interpreting this value.

IOCTL_PCI_ALT_PUT_TX_DATA

Function: Load a Tx data byte.
Input: TX_DATA_LOAD structure
Output: None
Notes: The TX_DATA_LOAD structure has two eight-bit fields: Channel – the number of the single transmit FIFO to write to, and Data – the data byte to write. This command is used when a small amount of data is to be transferred.

IOCTL_PCI_ALT_GET_RX_DATA

Function: Read an Rx data byte.
Input: Unsigned character
Output: Unsigned character
Notes: The number of the receive FIFO to read from is passed to this command and a byte of data read from the specified channel's FIFO is returned. This command is used when a small amount of data is to be transferred.

IOCTL_PCI_ALT_PUT_DMA_DATA

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

IOCTL_PCI_ALT_GET_DMA_DATA

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

IOCTL_PCI_ALT_RESET_FIFOS

Function: Reset the transmit and/or DMA FIFOs.
Input: FIFO_RESET structure or FIFO_SEL enumeration type
Output: None
Notes: This ioctl takes either the FIFO_SEL enumeration type or the FIFO_RESET structure. With FIFO_SEL the ioctl resets either the DMA FIFO, all eight transmit FIFOs, or both depending on the input value. This input was retained to maintain backward compatibility. The FIFO_RESET structure allows the transmit FIFOs to be individually reset. This capability was added with the rev. E board when separate reset and programmable flag load lines were added for each transmit and receive FIFO.

IOCTL_PCI_ALT_SET_TX_LEVEL

Function: Set the programmable almost empty level for a transmit FIFO.
Input: FIFO_LEVEL_LOAD structure
Output: None
Notes: Determines the number of FIFO bytes above empty that the programmable almost empty status will become true. See DDPciAlt.h for the definition of FIFO_LEVEL_LOAD.

IOCTL_PCI_ALT_GET_TX_LEVEL

Function: Return the programmable almost empty level for a transmit FIFO.
Input: Unsigned character (FIFO number)
Output: Unsigned short integer
Notes: Returns the number of FIFO bytes above empty that the programmable almost empty status will become true.

IOCTL_PCI_ALT_SET_RX_LEVEL

Function: Set the programmable almost full level for a receive FIFO.
Input: FIFO_LEVEL_LOAD structure
Output: None
Notes: Determines the number of FIFO bytes below full that the programmable almost full status will become true. See DDPciAlt.h for the definition of FIFO_LEVEL_LOAD.

IOCTL_PCI_ALT_GET_RX_LEVEL

Function: Return the programmable almost full level for a receive FIFO.
Input: Unsigned character (FIFO number)
Output: Unsigned short integer
Notes: Returns the number of FIFO bytes below full that the programmable almost full status will become true.

IOCTL_PCI_ALT_GET_ISR_STATUS

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

IOCTL_PCI_ALT_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 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_PCI_ALT_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_PCI_ALT_DISABLE_INTERRUPT

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

IOCTL_PCI_ALT_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_PCI_ALT_LOAD_ALTERA

Function: Loads a new configuration file to the Altera FPGA.
Input: ALTERA_LOAD structure
Output: None
Notes: The ALTERA_LOAD structure contains one field: an array of Unicode characters that specifies the file name of the file to load. The file name cannot exceed 32 characters including the file extension and terminating null character.

Write

PCI-Altera 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 size of that buffer in bytes, 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-Altera 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 size of that buffer in bytes, 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.

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
150 DuBois, Suite 3
Santa Cruz, CA 95060
(831) 457-8891 - Fax (831) 457-4793
support@dyneng.com