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

User Manual

PCI-Altera-485

Driver Documentation
CSC Version

Revision A1
Corresponding Hardware: Revision D
10-2002-0704
Corresponding Firmware: Revision G

PCI-Altera-485
PCI based Re-configurable logic with RS-485 and TTL IO

Dynamic Engineering
150 DuBois St. Suite 3
Santa Cruz, CA 95060
831- 457-8891
831-457-4793 FAX

Manual Revision A1. Revised December 10, 2007

Table of Contents

Introduction

The PciAltera driver is a WindowsXP driver for the PciAltera from Dynamic Engineering. Each PciAltera board. When the PciAltera is recognized by the PCI bus configuration. If the device has been previously installed and its device driver is loaded, a Device Object will be created. A separate handle to the PciAltera can be obtained using CreateFile() calls (see below). IO Control calls (IOCTLs) are used to configure the PciAltera and read status. The PciAltera is responsible for reading its user switch setting, operating the onboard LEDs, and a few other operations.

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-485 device user manual.

Driver Installation

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

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

The PATest.exe file is a sample WindowsXP console application that makes calls into the PciAlt 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 ***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.

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 PciAltGUID.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                  hPci3ip  = 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_PCI3IP,
                          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_PCI3IP,
                          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
hPci3ip  = CreateFile(deviceName,
                  GENERIC_READ    | GENERIC_WRITE,
                  FILE_SHARE_READ | FILE_SHARE_WRITE,
                  NULL,
                  OPEN_EXISTING,
                  NULL,
                  NULL);

if(hPci3ip  == 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.

```
 // Set the Altera Base configuration
 // Input:  Value of Altera Base control configuration (ULONG)
 // Output: None
#define IOCTL_PCIALT_SET_ALT_CONFIG\
        PCIALT_MAKE_IOCTL(PCIALT_DEVICE_TYPE, 20)

 // Return the Altera Base configuration
 // Input:  None
 // Output: Value of Altera Base control register (ULONG)
#define IOCTL_PCIALT_GET_ALT_CONFIG\
        PCIALT_MAKE_IOCTL(PCIALT_DEVICE_TYPE, 21)

 // Load the Altera with the specified configuration file
 // Input:  File name info for configuration file
 // Output: None
#define IOCTL_PCIALT_LOAD_ALTERA\
        PCIALT_MAKE_IOCTL(PCIALT_DEVICE_TYPE, 22)


 // Structures for IOCTLs
#define NUM_CHANNELS          8
#define FILE_NAME_SZ         32

 // Driver/Device information
typedef struct _DRIVER_DEVICE_INFO
{
   UCHAR    DriverVersion;
   UCHAR    SwitchValue;
   ULONG    InstanceNumber;
   USHORT   TxFifoSizes[NUM_CHANNELS];
   USHORT   RxFifoSizes[NUM_CHANNELS];
} DRIVER_DEVICE_INFO, *PDRIVER_DEVICE_INFO;
```

```
 // Command information structure
typedef struct _TRANSFER_COMMAND
{
   BOOLEAN  Transmit;
   UCHAR    Channels;
   USHORT   Count;
} TRANSFER_COMMAND, *PTRANSFER_COMMAND;

 // Command status and count of commands pending
typedef struct _COMMAND_STAT
{
   ULONG    Status;
   UCHAR    Count;
} COMMAND_STAT, *PCOMMAND_STAT;

typedef struct _TX_DATA_LOAD
{
   UCHAR    Channel;
   UCHAR    Data;
} TX_DATA_LOAD, *PTX_DATA_LOAD;

typedef struct _ALTERA_LOAD
{
   WCHAR    FileName[FILE_NAME_SZ];
} ALTERA_LOAD, *PALTERA_LOAD;

typedef enum _FIFO_SEL {TX, DMA, BOTH} FIFO_SEL, *PFIFO_SEL;
```

IOCTL_PCIALT_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 dip-switch 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_PCIALT_GET_STATUS

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

IOCTL_PCIALT_SET_CONFIG

Function: Writes to the base configuration register on the PCI-Altera-485.
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 PciAltDef.h for information on determining this value.

IOCTL_PCIALT_GET_CONFIG

Function: Returns the configuration of the base control register.
Input: none
Output: Unsigned long int.
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_PCIALT_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_PCIALT_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 PciAltDef.h for information on interpreting this value.

IOCTL_PCIALT_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_PCIALT_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 PciAltDef.h for information on determining this value.

IOCTL_PCIALT_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 PciAltDef.h for information on interpreting this value.

IOCTL_PCIALT_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 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 PciAltDef.h for information on interpreting this value.

IOCTL_PCIALT_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_PCIALT_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_PCIALT_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_PCIALT_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_PCIALT_RESET_FIFOS

Function: Reset the transmit and/or DMA FIFOs.
Input: FIFO_SEL enumeration type.
Output: none
Notes: Resets either the DMA FIFO, all eight transmit FIFOs, or both depending on the input value.

IOCTL_PCIALT_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_PCIALT_SET_INT_CONFIG command. The interrupts that deal with the DMA transfers do not affect the value.

IOCTL_PCIALT_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_PCIALT_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_PCIALT_DISABLE_INTERRUPT

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

IOCTL_PCIALT_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_PCIALT_ENABLE_RIF_INTERRUPT

Function: Enable Rif interrupt in Lif control/status register.
Input: none
Output: none
Notes:.

IOCTL_PCIALT_DISABLE_RIF_INTERRUPT

Function: Disable Rif interrupt in Lif control/status register.
Input: none
Output: none
Notes:.

IOCTL_PCIALT_GET_LIF_STATUS

Function: Read Lif status.
Input: none

Output: USHORT
Notes:.

IOCTL_PCIALT_GET_BUS_ERROR_STATUS

Function: Read and clear bus error status from RIF I/F read timeout.
Input: none
Output: USHORT
Notes:.

IOCTL_PCIALT_WRITE_RIF_DATA

Function: Write data to the RIF.
Input: RIF_WRITE struct
Output: none
Notes:.

IOCTL_PCIALT_READ_RIF_DATA

Function: Read Rif data.
Input: RIF_READ struct
Output: USHORT
Notes:.

IOCTL_PCIALT_SET_ALT_CONFIG

Function: Set the Altera Base configuration.
Input: Value of Altera Base control configuration (ULONG)
Output: None
Notes:.

IOCTL_PCIALT_GET_ALT_CONFIG

Function: Return the Altera Base configuration.
Input: none
Output: Value of Altera Base control register (ULONG)
Notes:.

IOCTL_PCIALT_GET_LAST_RIF_ADDRESS

Function: Return the address of the last RIF access.
Input: none
Output: USHORT
Notes:.

```
 // Driver/Device information
typedef struct _DRIVER_DEVICE_INFO
{
   UCHAR    DriverVersion;
   UCHAR    SwitchValue;
   ULONG    InstanceNumber;
} DRIVER_DEVICE_INFO, *PDRIVER_DEVICE_INFO;

 // Command information structure
typedef struct _TRANSFER_COMMAND
{
   BOOLEAN  Transmit;
   UCHAR    Channels;
   USHORT   Count;
} TRANSFER_COMMAND, *PTRANSFER_COMMAND;

 // Command status and count of commands pending
typedef struct _COMMAND_STAT
{
   ULONG    Status;
   UCHAR    Count;
} COMMAND_STAT, *PCOMMAND_STAT;

typedef struct _TX_DATA_LOAD
{
   UCHAR    Channel;
   UCHAR    Data;
} TX_DATA_LOAD, *PTX_DATA_LOAD;

typedef enum _FIFO_SEL {TX, DMA, BOTH} FIFO_SEL, *PFIFO_SEL;

 // Altera Rif Write Address, Data, Flag
typedef struct _RIF_WRITE
{
```

```
   USHORT   Address;
   USHORT   Data;
   BOOLEAN  ChanAccess;
} RIF_WRITE, *PRIF_WRITE;

 // Altera Rif Read Address/Flag
typedef struct _RIF_READ
{
   USHORT   Address;
   BOOLEAN  ChanAccess;
} RIF_READ, *PRIF_READ;
```

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 St. Suite 3
Santa Cruz, CA 95060
831-457-8891
831-457-4793 fax
support@dyneng.com