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

PciAlt
AltAtp
AltEio
AltGen

WDF Device Drivers for the PCI-Altera

Driver Documentation

Windows Driver Foundation

Revision A
Corresponding Hardware: Revision
10-2002-0708
Corresponding Firmware: Revision L

PciAlt, AltAtp, AltEio and AltGen
WDF Device Drivers for the
PCI-Altera - PCI based interface module
With Re-programmable I/O logic

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

Manual Revision A. Revised February 1, 2016

Table of Contents

Introduction

The PciAlt, AltAtp, AltEio and AltGen drivers are Windows device drivers for the PCI-Altera-485/LVDS from Dynamic Engineering. These drivers were developed with the Windows Driver Foundation version 1.9 (WDF) from Microsoft, specifically the Kernel-Mode Driver Framework (KMDF).

The PCI-Altera board has a PLX PCI-9054 to implement the PCI interface with two DMA channels, one for DMA from the board to system memory and the other for the opposite transfer. A Xilinx XC2S100 and an Altera EP20K400E FPGA are mapped to the PLX local address space. DMA data is routed through the Xilinx to sixteen I/O FIFOs (eight input and eight output) that transfer data between the Xilinx and Altera to implement eight full-duplex I/O channels.

The Altera FPGA may be loaded by an optional onboard Flash memory, or with a bit-file transferred to the device via the PCI bus. The Altera controls 40 RS-485 or LVDS transceivers and 12 TTL bidirectional 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.

When the PCI-Altera is recognized by the PCI bus configuration utility it will start the PciAlt driver, which reads the registry to obtain the default Altera configuration file and path and programs the Altera FPGA if the file is present. The location and name of this configuration file is defined in the PciAlt.inf file; \SystemRoot\AlteraDesigns\AltATP.rbf is the default location and file name. SystemRoot is the Windows directory on the host machine. 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 design ID value is read from the Altera base address and the appropriate Altera driver is loaded to control the various Altera functions. 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 old Altera driver will then be unloaded and the new Altera driver will be loaded automatically if it has been previously installed. Separate handles to the PciAlt and to the current Altera driver can be obtained using CreateFile() calls (see example UserApp code). 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 of the board.

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/LVDS user manual (also referred to as the hardware manual).

Driver Installation

There are several files provided in each driver package. The files needed to install the PciAlt driver are PciAlt.inf, PciAlt.cat, PciAlt.sys and WdfCoInstaller01009.dll. The Altera driver files needed for installation are AlteraDesigns.inf, AlteraDesigns.cat, AltAtp.sys, AltEio.sys, AltGen.sys as well as WdfCoInstaller01009.dll.

Once the PciAlt driver is installed and the Altera is programmed, an Altera driver can be installed. The AltAtp, AltEio and AltGen drivers are included with the driver package. The AltAtp and AltEio are used by Dynamic Engineering to test the PCI-Altera board. The AltAtp driver is loaded for design ID zero and checks all the basic functions of the board. The AltEio is used to run enhanced I/O tests checking each I/O line as both an input and output at a high bit-rate. Six Altera designs are used with the AltEio driver with design IDs one through six. Designs one through five have eight I/O channels each using four of the 40 LVDS/RS485 I/O lines. The sixth design has six I/O channels each using four of the twelve TTL I/O lines. All I/O lines are tested as both an output and an input.

PciAltPublic.h, AltAtpPublic.h, AltEioPublic.h and AltGenPublic.h are C header files that define the Application Program Interface (API) to the drivers. These files are required at compile time by any application that wishes to interface with the drivers, but they are not needed for driver installation.

Note: In order to build an application you must link with setupapi.lib.

Windows 7 Installation

Copy PciAlt.inf, PciAlt.cat, PciAlt.sys and WdfCoInstaller01009.dll (Win7 version) to a CD, USB memory device or other system accessible location as preferred.

With the PCI-Altera hardware installed, power-on the PCI host computer.

- Open the ***Device Manager*** from the control panel.
- Under ***Other devices*** there should be an ***Other PCI Bridge Device****.
- Right-click on the ***Other PCI Bridge Device*** and select ***Update Driver Software***.
- Insert the disk or memory device prepared above in the desired drive.
- Select ***Browse my computer for driver software***.
- Enter the path prepared above ***Search for driver software in this location*** or use the Browse option to navigate to the proper location.
- Select ***Next***.
- Select ***Close*** to close the update window.

  The system should now display the PciAlt I/O channel in the Device Manager.

- Right-click on the device icon, select ***Update Driver Software*** and proceed as above. If the Altera is not seen it may be necessary to restart the host computer to load registry information.

***** If the ***Other PCI Bridge Device*** is not displayed, click on the ***Scan for hardware changes*** icon on the tool-bar.

Driver Startup

Once the drivers have been installed they will start automatically when the system recognizes the hardware.

Handles can be opened to a specific board by using the CreateFile() function call and passing in the device names obtained from the system.

The interfaces to the devices are identified using globally unique identifiers (GUID), which are defined in PciAltPublic.h, AltAtpPublic.h, AltEioPublic.h and AltGenPublic.h. See main.c in the PciAlteraUserApp project for an example of how to acquire handles for the PciAlt and Altera devices.

IO Controls

The drivers use IO Control calls (IOCTLs) to configure the devices. IOCTLs refer to a single Device Object, which controls a single board or I/O channel. IOCTLs are called using the Win32 function DeviceIoControl() (see below), and passing in the handle to the device opened with CreateFile() (see above). 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 for the PciAlt driver are described below:

IOCTL_PCI_ALT_GET_INFO
Function: Returns the Driver revision, Xilinx flash revision, Switch value, Instance Number, and Transmit and Receive FIFO sizes.
Input: None
Output: PCI_ALT_DRIVER_DEVICE_INFO structure
Notes: Switch value is the configuration of the eight-position onboard dipswitch that has been selected by the User (see the board silk screen for bit position and polarity). Instance Number is the zero-based order in which the system initializes the PciAlt devices. The FIFO sizes are dynamically detected when the driver starts up. See the definition of PCI_ALT_DRIVER_DEVICE_INFO below.

```
#define NUM_CHANNELS             8

 // Driver/Device information
typedef struct _PCI_ALT_DRIVER_DEVICE_INFO {
   UCHAR    DriverRev;
   UCHAR    XilinxRev;
   UCHAR    SwitchValue;
   ULONG    InstanceNum;
   USHORT   TxFifoSizes[NUM_CHANNELS];
   USHORT   RxFifoSizes[NUM_CHANNELS];
} PCI_ALT_DRIVER_DEVICE_INFO, *PPCI_ALT_DRIVER_DEVICE_INFO;
```

IOCTL_PCI_ALT_GET_STATUS
Function: Returns the FIFO data levels and other status information.
Input: None
Output: BASE_STAT structure
Notes: See the definition of BASE_STAT below.

```
 // Base status structure
typedef struct _BASE_STAT {
   UCHAR    RxFfAFl;    // Mask of receive FIFO almost full channels
   UCHAR    TxFfAMt;    // Mask of transmit FIFO almost empty channels
   BOOLEAN  DmaFfMt;    // DMA FIFO empty flag
   BOOLEAN  DmaFfFl;    // DMA FIFO full flag
   BOOLEAN  DmaFfVld;   // DMA FIFO data valid flag
   BOOLEAN  AltDone;    // Altera done flag
   BOOLEAN  AltStat;    // Altera nStatus value
   BOOLEAN  AXLink;     // AXLink input value
   BOOLEAN  LocInt;     // Local interrupt active
} BASE_STAT, *PBASE_STAT;
```

IOCTL_PCI_ALT_SET_CONFIG
Function: Writes to the base configuration register on the PCI-Altera.
Input: BASE_CONFIG structure
Output: None
Notes: Controls the single Xilinx LED and the direction and data value when configured as an output of the XALink line. A bidirectional line between the Xilinx and the Altera devices. See the definition of BASE_CONFIG below.

```
 // Base configuration structure
typedef struct _BASE_CONFIG {
   BOOLEAN  XLedOn;     // When TRUE turn XLed on
   BOOLEAN  XALinkDir;  // When TRUE XA link line is an output
   BOOLEAN  XALinkData; // XA Link value when configured as output
} BASE_CONFIG, *PBASE_CONFIG;
```

IOCTL_PCI_ALT_GET_CONFIG
Function: Returns the configuration of the base control register.
Input: None
Output: BASE_CONFIG structure
Notes: This command is used mainly for testing to verify the set command or to store the current configuration to be restored later.

IOCTL_PCI_ALT_LOAD_COMMAND
Function: Loads an I/O data transfer command into the eight-position command queue.
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).
CmndDir – true for a transmit command, false for a receive command.

```
 // Command information structure
typedef struct _TRANSFER_COMMAND {
   BOOLEAN  CmndDir;                // True: Xil->Alt False: Alt->Xil
   BOOLEAN  ActvChans[NUM_CHANNELS];// Target channel(s) for this command
   USHORT   WordCount;              // Number of 32-bit words to transfer
} TRANSFER_COMMAND, *PTRANSFER_COMMAND;
```

IOCTL_PCI_ALT_GET_COMMAND_STAT
Function: Returns the command status and command queue count.
Input: None
Output: COMMAND_STAT structure
Notes: The COMMAND_STAT structure has seven fields: CmndCount is the number of commands currently in the command queue, CmndActv is true if a command is currently running, CmndDir is true for a transmit command, false for a receive command, CmndReady is true if the command queue is not empty, CmndFull is true if the eight-position command queue is full. FifoCount is the number of 32-bit words in the DMA FIFO and ActvChans is a mask of the active channels for the current or latest command. See the definition of COMMAND_STAT below.

```
 // Command status and count of commands pending
typedef struct _COMMAND_STAT {
   UCHAR    CmndCount;  // Number of commands pending
   BOOLEAN  CmndActv;   // True when a command is currently executing
   BOOLEAN  CmndDir;    // True: Xil->Alt False: Alt->Xil
   BOOLEAN  CmndReady;  // True if command queue is not empty
   BOOLEAN  CmndFull;   // True if command queue is full
   USHORT   FifoCount;  // Number of words in the DMA FIFO
   UCHAR    ActvChans;  // Active command target channels
} COMMAND_STAT, *PCOMMAND_STAT;
```

IOCTL_PCI_ALT_RESET_COMMAND_QUEUE
Function: Resets 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: Sets the Interrupt enable configuration.
Input: INT_CONFIG structure
Output: None
Notes: This command determines which conditions are enabled to cause an interrupt when the master interrupt enable is set. See the bit definition of INT_CONFIG below.

```
typedef struct _INT_CONFIG {
   BOOLEAN  AlteraDone;     // Altera configuration done interrupt
   BOOLEAN  AltXilLink;     // Altera->Xilinx link interrupt (active high)
   BOOLEAN  CmndDone;       // Command done interrupt
   BOOLEAN  TransmitDone;   // Xilinx->Altera transfer complete interrupt
   BOOLEAN  ReceiveDone;    // Altera->Xilinx transfer complete interrupt
   BOOLEAN  TxAmtInt[NUM_CHANNELS]; // Tx FIFO almost empty interrupts
   BOOLEAN  RxAflInt[NUM_CHANNELS]; // Rx FIFO almost full interrupts
} INT_CONFIG, *PINT_CONFIG;
```

IOCTL_PCI_ALT_GET_INT_CONFIG
Function: Returns the Interrupt enable configuration.
Input: None
Output: INT_CONFIG structure
Notes: Returns the signals enabled to cause an interrupt. See the bit definition of INT_CONFIG above.

IOCTL_PCI_ALT_GET_INT_STAT
Function: Returns 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 bits below for information on interpreting this value.

```
 // Interrupt latched status bits
#define INTSTAT_RX_FIFO_0_AF      0x00000001  // Rx FIFO 0 almost full int
#define INTSTAT_RX_FIFO_1_AF      0x00000002  // Rx FIFO 1 almost full int
#define INTSTAT_RX_FIFO_2_AF      0x00000004  // Rx FIFO 2 almost full int
#define INTSTAT_RX_FIFO_3_AF      0x00000008  // Rx FIFO 3 almost full int
#define INTSTAT_RX_FIFO_4_AF      0x00000010  // Rx FIFO 4 almost full int
#define INTSTAT_RX_FIFO_5_AF      0x00000020  // Rx FIFO 5 almost full int
#define INTSTAT_RX_FIFO_6_AF      0x00000040  // Rx FIFO 6 almost full int
#define INTSTAT_RX_FIFO_7_AF      0x00000080  // Rx FIFO 7 almost full int
#define INTSTAT_TX_FIFO_0_AE      0x00000100  // Tx FIFO 0 almost empty int
#define INTSTAT_TX_FIFO_1_AE      0x00000200  // Tx FIFO 1 almost empty int
#define INTSTAT_TX_FIFO_2_AE      0x00000400  // Tx FIFO 2 almost empty int
#define INTSTAT_TX_FIFO_3_AE      0x00000800  // Tx FIFO 3 almost empty int
#define INTSTAT_TX_FIFO_4_AE      0x00001000  // Tx FIFO 4 almost empty int
#define INTSTAT_TX_FIFO_5_AE      0x00002000  // Tx FIFO 5 almost empty int
#define INTSTAT_TX_FIFO_6_AE      0x00004000  // Tx FIFO 6 almost empty int
#define INTSTAT_TX_FIFO_7_AE      0x00008000  // Tx FIFO 7 almost empty int
#define INTSTAT_ALTERA_DONE       0x00010000  // Altera done configuring int
#define INTSTAT_ALT_XIL_LINK      0x00020000  // Altera->Xilinx link int
#define INTSTAT_COMMAND_DONE      0x00040000  // Command done int
#define INTSTAT_TX_WRITE_DONE     0x00080000  // Tx write done int
#define INTSTAT_RX_READ_DONE      0x00100000  // Rx read done int
#define INTSTAT_INT_ACTIVE        0x01000000  // Enabled interrupt active
```

IOCTL_PCI_ALT_PUT_TX_DATA

Function: Loads 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 can be used when a small amount of data is to be transferred.

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

IOCTL_PCI_ALT_GET_RX_DATA

Function: Reads 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 can be used when a small amount of data is to be read.

IOCTL_PCI_ALT_PUT_DMA_DATA
Function: Loads a 32-bit 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: Reads a 32-bit 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: Resets the transmit and/or DMA FIFOs.
Input: FIFO_SEL enumerated type or FIFO_RESET structure
Output: None
Notes: This call accepts either the FIFO_SEL enumerated type or the FIFO_RESET structure. With FIFO_SEL the call 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.

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

typedef struct _FIFO_RESET {
   BOOLEAN  ResetDMA;
   UCHAR    TxResetMask;
} FIFO_RESET, *PFIFO_RESET;
```

IOCTL_PCI_ALT_SET_TX_LEVEL
Function: Sets 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 the definition of FIFO_LEVEL_LOAD below.

```
typedef struct _FIFO_LEVEL_LOAD {
   UCHAR    Channel;
   USHORT   Level;
} FIFO_LEVEL_LOAD, *PFIFO_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: Sets 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 the definition of FIFO_LEVEL_LOAD above.

IOCTL_PCI_ALT_GET_RX_LEVEL
Function: Returns 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: Returns the interrupt status read in the ISR from the last interrupt.
Input: None
Output: PCI_ALT_ISR_STAT structure
Notes: The Status 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 this value. New will be true if a user interrupt has occurred since the ISR was last read.

```
typedef struct _PCI_ALT_ISR_STAT {
   ULONG    Status;  // Value of status register read in ISR
   BOOLEAN  New;     // True if the status has changed since last read
} PCI_ALT_ISR_STAT, *PPCI_ALT_ISR_STAT;
```

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: 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 after each user interrupt to re-enable it.

IOCTL_PCI_ALT_DISABLE_INTERRUPT
Function: Disables 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. All configuration files must be stored in the AlteraDesigns folder under the Windows directory. The file name cannot exceed 32 characters including the file extension and terminating null character.

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

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.

The IOCTLs defined for the AltAtp driver are described below:

IOCTL_ALT_ATP_GET_INFO
Function: Returns the Driver Revision, Design ID, Instance Number and the device IDs of the eight PLL devices.
Input: None
Output: DRIVER_ALT_DEVICE_INFO structure
Notes: Design ID is read from the Altera Base register, Instance Number is the zero-based order in which the system initializes the AltAtp devices, PLL device IDs are dynamically detected when the driver starts up. They can be one of two values: 0x69 or 0x6A.

```
 //  Driver revision and instance information
typedef struct _DRIVER_ALT_DEVICE_INFO {
   UCHAR    DriverRev;
   UCHAR    DesignId;
   ULONG    InstanceNum;
   UCHAR    PllDevIds[NUM_PLLS];
} DRIVER_ALT_DEVICE_INFO, *PDRIVER_ALT_DEVICE_INFO;
```

IOCTL_ALT_ATP_SET_LEDS
Function: Controls the state of the four Altera LEDs – A_Led0-3 on the upper right-hand corner of the board.
Input: Unsigned character (LED configuration)
Output: None
Notes: A value of zero turns off all four LEDs. A one in a bit position 0..3 turns on the corresponding LED.

IOCTL_ALT_ATP_SET_CONFIG
Function: Sets the start bits for the receiver and transmitter state machines, the AX link signal direction and output data value and the XALink input data value.
Input: RXTX_CONFIG structure
Output: None
Notes: See the definition of RXTX_CONFIG below.

```
typedef struct _RXTX_CONFIG {
   UCHAR    RxStartChans;  // Enabled receiver channel mask
   UCHAR    TxStartChans;  // Enabled transmitter channel mask
   BOOLEAN  AXLinkDir;     // When TRUE AX link line is an output
   BOOLEAN  AXLinkData;    // AX Link value when configured as output
   BOOLEAN  XALinkDatIn;   // Input link data from Xilinx
} RXTX_CONFIG, *PRXTX_CONFIG;
```

IOCTL_ALT_ATP_GET_CONFIG
Function: Returns the configuration of the I/O control register and the XALink line input data value.
Input: None
Output: RXTX_CONFIG structure
Notes: See the definition of RXTX_CONFIG above.

IOCTL_ALT_ATP_SET_DIR
Function: Sets the direction of the 40 RS-485/LVDS IO lines.
Input: DTIO_BITS structure
Output: None
Notes: The input structure contains two long words; each controls 20 IO lines. See the definition of DTIO_BITS below.

```
typedef struct _DTIO_BITS {
   ULONG    Lower20;
   ULONG    Upper20;
} DTIO_BITS, *PDTIO_BITS;
```

IOCTL_ALT_ATP_GET_DIR
Function: Returns the direction of the 40 RS-485/LVDS IO lines.
Input: None
Output: DTIO_BITS structure
Notes: See the definition of DTIO_BITS above.

IOCTL_ALT_ATP_SET_TERM
Function: Sets the terminations on the 40 RS-485/LVDS IO lines.
Input: DTIO_BITS structure
Output: None
Notes: See the definition of DTIO_BITS above.

IOCTL_ALT_ATP_GET_TERM
Function: Returns the active terminations on the 40 RS-485/LVDS IO lines.
Input: None
Output: DTIO_BITS structure
Notes: See the definition of DTIO_BITS above.

IOCTL_ALT_ATP_SET_IO_DATA
Function: Sets the data-values driven onto the 40 RS-485/LVDS IO lines when they are configured as outputs.
Input: DTIO_BITS structure
Output: None
Notes: See the definition of DTIO_BITS above.

IOCTL_ALT_ATP_GET_IO_DATA
Function: Returns the data-values read from the 40 RS-485/LVDS IO lines.
Input: None
Output: DTIO_BITS structure
Notes: See the definition of DTIO_BITS above.

IOCTL_ALT_ATP_SET_TTL_DATA
Function: Sets the values of the 12 TTL data-lines.
Input: Unsigned short integer (TTL output data-bits 0-11)
Output: None
Notes: When the output drivers are disabled, the level of the external lines can be read from the external TTL lines. The active low TTL driver enables are tied to the data bits, therefore the data value must be set high in order for the line to be used as an input.

IOCTL_ALT_ATP_GET_TTL_DATA
Function: Returns the values read from the 12 TTL data-lines.
Input: None
Output: Unsigned short integer (TTL input data-bits 0-11)
Notes: The active low TTL driver enables are tied to the data output bits, therefore the data value must be set high in order for the line to be used as an input, otherwise a low will be read regardless of the input level.

IOCTL_ALT_ATP_PUT_RX_DATA
Function: Loads an Rx data byte to a single channel's receive FIFO.
Input: RX_DATA_LOAD structure
Output: None
Notes: The RX_DATA_LOAD structure has two eight-bit fields: Channel – the number of the single receive FIFO to write to, and Data – the data byte to write. See the definition of RX_DATA_LOAD below.

```
typedef struct _RX_DATA_LOAD {
   UCHAR    Channel;
   UCHAR    Data;
} RX_DATA_LOAD, *PRX_DATA_LOAD;
```

IOCTL_ALT_ATP_GET_TX_DATA
Function: Reads a data-byte from a channel's transmit FIFO.
Input: Unsigned character (channel number)
Output: Unsigned character (transmit data value)
Notes: The number of the transmit FIFO to read from is passed to this command and a byte of data read from the specified channel's FIFO is returned

IOCTL_ALT_ATP_RESET_RX_FIFOS
Function: Resets the Rx FIFOs.
Input: None
Output: None
Notes: Resets all eight receive FIFOs.

IOCTL_ALT_ATP_SET_RX_LEVEL
Function: Sets an Rx FIFO's almost full level.
Input: RX_LEVEL_LOAD structure
Output: None
Notes: The RX_LEVEL_LOAD structure has an eight-bit field: Channel – the number of the single receive FIFO to write to, and a 16-bit field Data – the almost full level to write. See the definition of RX_LEVEL_LOAD below.

```
typedef struct _RX_LEVEL_LOAD {
   UCHAR    Channel;
   USHORT   Data;
} RX_LEVEL_LOAD, *PRX_LEVEL_LOAD;
```

IOCTL_ALT_ATP_GET_FIFO_STATUS
Function: Returns all the receiver and transmitter FIFO level flags.
Input: None
Output: FIFO_STATUS structure
Notes: Each of the six fields is a bit-mask of the channels for which the respective FIFO flags are true. See the definition of FIFO_STATUS below.

```
typedef struct _FIFO_STATUS {
   UCHAR    RxEmpty;
   UCHAR    RxFull;
   UCHAR    TxEmpty;
   UCHAR    TxFull;
   UCHAR    RxAlmostFull;
   UCHAR    TxAlmostEmpty;
} FIFO_STATUS, *PFIFO_STATUS;
```

IOCTL_ALT_ATP_READ_PLL_DATA
Function: Returns the contents of a single PLL's internal registers.
Input: Unsigned character (PLL number (0-7))
Output: PLL_READ structure
Notes: The channel number of the PLL to read from is passed in to this call and the register data is output in the PLL_READ struct in an array or 40 bytes. If channel is greater than seven, the first byte of the data array will return the value of the S2 bits from the eight PLLs. See the definition of PLL_READ below.

```
typedef struct _PLL_READ {
   UCHAR   Data[PLL_MESSAGE_SIZE];
} PLL_READ, *PPLL_READ;
```

IOCTL_ALT_ATP_LOAD_PLL_DATA
Function: Loads the internal registers of a single PLL.
Input: PLL_LOAD structure
Output: None
Notes: The PLL_LOAD structure has two fields: Channel – the number of the PLL to write to, and Data – an array of 40 bytes containing the data to write. If channel is greater than seven, the first byte of data is written to the S2 bits for the eight PLLs. See the definition of PLL_LOAD below.

```
typedef struct _PLL_LOAD {
   UCHAR    Channel;
   UCHAR    Data[PLL_MESSAGE_SIZE];
} PLL_LOAD, *PPLL_LOAD;
```

IOCTL_ALT_ATP_SET_OSC_CONTROL
Function: Configures the oscillator counters and outputs.
Input: OSC_CONTROL structure
Output: None
Notes: The OSC_CONTROL structure has four fields: OutEnables individually enables the 24 PLL inputs onto IO 0..23 when these lines are configured as outputs. CountClear clears the counts of the 25 counters that count at the rates of the 24 PLL clock inputs and the reference oscillator. CountEnable enables all the counters to count until the master counter (clocked by the reference oscillator) reaches a count of 0x1000000. At this point all counters stop counting and their counts can be read to verify the frequencies of the various PLL outputs. MuxSelect selects the counter that will be read with the next IOCTL_ALT_ATP_READ_OSC_DATA call. See the definition of OSC_CONTROL below.

```
typedef struct _OSC_CONTROL {
   ULONG    OutEnables;
   USHORT   MuxSelect;
   BOOLEAN  CountEnable;
   BOOLEAN  CountClear;
} OSC_CONTROL, *POSC_CONTROL;
```

IOCTL_ALT_ATP_GET_OSC_CONTROL
Function: Returns the current oscillator counters and outputs configuration.
Input: None
Output: OSC_CONTROL structure
Notes: See the definition of OSC_CONTROL above.

IOCTL_ALT_ATP_READ_OSC_DATA
Function: Reads the count of the counter specified in the last IOCTL_ALT_ATP_SET_OSC_CONTROL call.
Input: None
Output: Unsigned long integer (counter value)
Notes: The frequency of the PLL selected can be calculated by the following formula: F = 66.6667 MHz * count / 16,777,216.

IOCTL_ALT_ATP_REGISTER_EVENT
Function: Registers an event to be signaled when a user 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.

IOCTL_ALT_ATP_ENABLE_INTERRUPT
Function: Enables the master interrupt.
Input: None
Output: None
Notes: This command must be run to allow the board to respond to interrupts. The master interrupt enable is disabled in the driver interrupt service routine. This command must then be run after each user interrupt to re-enable it.

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

IOCTL_ALT_ATP_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_ALT_ATP_PUT_DATA
Function: Writes one long word to the Altera memory space.
Input: ALT_DATA_LOAD structure
Output: None
Notes: The ALT_DATA_LOAD structure has two unsigned long integer fields: Address: the address offset value from the Altera base address, and Data: the data value to write to the above address. See the definition of ALT_DATA_LOAD below.

```
 // Load Altera data structure
typedef struct _ALT_DATA_LOAD {
   ULONG   Address;
   ULONG   Data;
} ALT_DATA_LOAD, *PALT_DATA_LOAD;
```

IOCTL_ALT_ATP_GET_DATA
Function: Reads one long word from the Altera memory space.
Input: Unsigned long integer (address offset)
Output: Unsigned long integer (data read)
Notes: As in the previous call the address offset value is passed into this call, but in this case the data value read from that address is returned.

The IOCTLs defined for the AltEio driver are described below:

IOCTL_ALT_EIO_GET_INFO
Function: Returns the Driver Revision, Design ID, Instance Number and the device IDs of the eight PLL devices.
Input: None
Output: DRIVER_ALT_DEVICE_INFO structure
Notes: Design ID is read from the Altera Base register, Instance Number is the zero-based order in which the system initializes the AltEio devices, PLL device IDs are dynamically detected when the driver starts up. They can be one of two values: 0x69 or 0x6A. See the definition of DRIVER_ALT_DEVICE_INFO below.

```
 // Driver/Device information
typedef struct _DRIVER_ALT_DEVICE_INFO {
   UCHAR    DriverRev;
   UCHAR    DesignId;
   ULONG    InstanceNum;
   UCHAR    PllDevIds[NUM_PLLS];
} DRIVER_ALT_DEVICE_INFO, *PDRIVER_ALT_DEVICE_INFO;
```

IOCTL_ALT_EIO_SET_LEDS
Function: Controls the state of the four LEDs – A_Led0-3 on the upper right-hand corner of the board.
Input: Unsigned character (LED configuration)
Output: None
Notes: A value of zero turns off all four LEDs. A one in a bit position 0..3 turns on the corresponding LED.

IOCTL_ALT_EIO_SET_CONFIG
Function: Sets the enable and start bits for the channel's I/O state machine and the AX link signal direction and output data value.
Input: CHAN_CONFIG structure
Output: None
Notes: See the definition of CHAN_CONFIG below.

```
typedef struct _CHAN_CONFIG {
   UCHAR    EnableChans;   // Link enabled channel mask
   UCHAR    StartChans;    // Link started channel mask
   BOOLEAN  AXLinkDir;     // When TRUE AX link line is an output
   BOOLEAN  AXLinkData;    // AX Link value when configured as output
   BOOLEAN  XALinkDatIn;   // Input link data from Xilinx
} CHAN_CONFIG, *PCHAN_CONFIG;
```

IOCTL_ALT_EIO_GET_CONFIG
Function: Returns the configuration of the I/O control register and the XALink line input data value.
Input: None
Output: CHAN_CONFIG structure
Notes: See the definition of CHAN_CONFIG above.

IOCTL_ALT_EIO_SET_COUNT
Function: Sets a channel's clock divider for the transmit bit-rate.
Input: CHAN_CLOCK_COUNT structure
Output: None
Notes: The count field determines the bit period of the transmit data stream. The value is one less than the number of clock periods in an output bit e.g. a value of zero results in a bit period of one clock period. See the definition of CHAN_CLOCK_COUNT below.

```
typedef struct _CHAN_CLOCK_COUNT {
   UCHAR    Channel;
   UCHAR    Count;       // 0->0xf => divide by 1->16
} CHAN_CLOCK_COUNT, *PCHAN_CLOCK_COUNT;
```

IOCTL_ALT_EIO_GET_COUNT
Function: Returns the value of a channel's transmit data clock divider.
Input: Unsigned character (channel number)
Output: Unsigned character (clock divider - 1)
Notes: The bit period of the transmit data stream can be determined by the return value. The value is one less than the number of clock periods in an output bit e.g. a value of zero specifies a bit period of one clock period.

IOCTL_ALT_EIO_PUT_RX_DATA
Function: Loads an Rx data byte to a single channel's receive FIFO.
Input: RX_DATA_LOAD structure
Output: None
Notes: The RX_DATA_LOAD structure has two eight-bit fields: Channel – the number of the single receive FIFO to write to, and Data – the data byte to write. See the definition of RX_DATA_LOAD below.

```
typedef struct _RX_DATA_LOAD {
   UCHAR    Channel;
   UCHAR    Data;
} RX_DATA_LOAD, *PRX_DATA_LOAD;
```

IOCTL_ALT_EIO_GET_TX_DATA
Function: Reads a data-byte from a channel's transmit FIFO.
Input: Unsigned character (channel number)
Output: Unsigned character (transmit data value)
Notes: The number of the transmit FIFO to read from is passed to this command and a byte of data read from the specified channel's FIFO is returned

IOCTL_ALT_EIO_RESET_RX_FIFOS
Function: Resets the Rx FIFOs.
Input: None
Output: None
Notes: Resets all eight receive FIFOs.

IOCTL_ALT_EIO_SET_RX_LEVEL
Function: Sets an Rx FIFO's almost full level.
Input: RX_LEVEL_LOAD structure
Output: None
Notes: The RX_LEVEL_LOAD structure has an eight-bit field: Channel – the number of the single receive FIFO to write to, and a 16-bit field Data – the almost full level to write. See the definition of RX_LEVEL_LOAD below.

```
typedef struct _RX_LEVEL_LOAD {
   UCHAR    Channel;
   USHORT   Data;
} RX_LEVEL_LOAD, *PRX_LEVEL_LOAD;
```

IOCTL_ALT_EIO_GET_FIFO_STATUS
Function: Returns all the receiver and transmitter FIFO level flags.
Input: None
Output: FIFO_STATUS structure
Notes: Each of the six fields is a bit-mask of the channels for which the respective FIFO flags are true. See the definition of FIFO_STATUS below.

```
typedef struct _FIFO_STATUS {
   UCHAR    RxEmpty;
   UCHAR    RxFull;
   UCHAR    TxEmpty;
   UCHAR    TxFull;
   UCHAR    RxAlmostFull;
   UCHAR    TxAlmostEmpty;
} FIFO_STATUS, *PFIFO_STATUS;
```

IOCTL_ALT_EIO_GET_LINK_STATUS
Function: Returns the link and transmit data status of the eight I/O channels.
Input: None
Output: LINK_STATUS structure
Notes: Each of the two fields is an eight-bit status mask of the eight I/O channels. When a channel is enabled and started, a link is established between the inputs and outputs of the channel and kept active by sending NULL characters whenever there is no data to transfer. The TxValid bits indicate that data is currently available to send. See the definition of LINK_STATUS below.

```
typedef struct _LINK_STATUS {
   UCHAR    TxValid;
   UCHAR    ChanLinked;
} LINK_STATUS, *PLINK_STATUS;
```

IOCTL_ALT_EIO_READ_PLL_DATA
Function: Returns the contents of a single PLL's internal registers.
Input: Unsigned character (PLL number (0-7))
Output: PLL_READ structure
Notes: The channel number of the PLL to read from is passed in to this call and the register data is output in the PLL_READ struct in an array or 40 bytes. If channel is greater than seven, the first byte of the data array will return the value of the S2 bits from the eight PLLs. See the definition of PLL_READ below.

```
typedef struct _PLL_READ {
   UCHAR   Data[PLL_MESSAGE_SIZE];
} PLL_READ, *PPLL_READ;
```

IOCTL_ALT_EIO_LOAD_PLL_DATA
Function: Loads the internal registers of a single PLL.
Input: PLL_LOAD structure
Output: None
Notes: The PLL_LOAD structure has two fields: Channel – the number of the PLL to write to, and Data – an array of 40 bytes containing the data to write. If channel is greater than seven, the first byte of data is written to the S2 bits for the eight PLLs. See the definition of PLL_LOAD below.

```
typedef struct _PLL_LOAD {
   UCHAR    Channel;
   UCHAR    Data[PLL_MESSAGE_SIZE];
} PLL_LOAD, *PPLL_LOAD;
```

IOCTL_ALT_EIO_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.

IOCTL_ALT_EIO_ENABLE_INTERRUPT
Function: Enables the master interrupt.
Input: None
Output: None
Notes: This command must be run to allow the board to respond to interrupts. The master interrupt enable is disabled in the driver interrupt service routine. This command must then be run again to re-enable it.

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

IOCTL_ALT_EIO_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.

The IOCTLs defined for the AltGen driver are described below:

IOCTL_ALT_GEN_GET_INFO
Function: Returns the Driver Revision, Design ID, Instance Number and the device IDs of the eight PLL devices.
Input: None
Output: DRIVER_ALT_DEVICE_INFO structure
Notes: Design ID is read from the Altera Base register, Instance Number is the zero-based order in which the system initializes the AltGen devices, PLL device IDs are dynamically detected when the driver starts up. They can be one of two values: 0x69 or 0x6A.

```
 // Driver/Device information
typedef struct _DRIVER_ALT_DEVICE_INFO {
   UCHAR    DriverRev;
   UCHAR    DesignId;
   ULONG    InstanceNum;
   UCHAR    PllDevIds[NUM_PLLS];
} DRIVER_ALT_DEVICE_INFO, *PDRIVER_ALT_DEVICE_INFO;
```

IOCTL_ALT_GEN_SET_LEDS
Function: Controls the state of the four LEDs – A_Led0-3 on the upper right-hand corner of the board.
Input: Unsigned character (LED configuration)
Output: None
Notes: A value of zero turns off all four LEDs. A one in a bit position 0..3 turns on the corresponding LED.

IOCTL_ALT_GEN_READ_PLL_DATA
Function: Returns the contents of a single PLL's internal registers.
Input: Unsigned character (PLL number (0-7))
Output: PLL_READ structure
Notes: The channel number of the PLL to read from is passed in to this call and the register data is output in the PLL_READ struct in an array or 40 bytes. If channel is greater than seven, the first byte of the data array will return the value of the S2 bits from the eight PLLs. See the definition of PLL_READ below.

```
typedef struct _PLL_READ {
   UCHAR   Data[PLL_MESSAGE_SIZE];
} PLL_READ, *PPLL_READ;
```

IOCTL_ALT_GEN_LOAD_PLL_DATA
Function: Loads the internal registers of a single PLL.
Input: PLL_LOAD structure
Output: None
Notes: The PLL_LOAD structure has two fields: Channel – the number of the PLL to write to, and Data – an array of 40 bytes containing the data to write. If channel is greater than seven, the first byte of data is written to the S2 bits for the eight PLLs. See the definition of PLL_LOAD below.

```
typedef struct _PLL_LOAD {
   UCHAR    Channel;
   UCHAR    Data[PLL_MESSAGE_SIZE];
} PLL_LOAD, *PPLL_LOAD;
```

IOCTL_ALT_GEN_PUT_DATA
Function: Writes one long word to the Altera memory space.
Input: ALT_DATA_LOAD structure
Output: None
Notes: The ALT_DATA_LOAD structure has two unsigned long int fields: Address: the address offset value from the Altera base address, and Data: the data value to write to the above address.

```
 // Load Altera data structure
typedef struct _ALT_DATA_LOAD {
   ULONG   Address;
   ULONG   Data;
} ALT_DATA_LOAD, *PALT_DATA_LOAD;
```

IOCTL_ALT_GEN_GET_DATA
Function: Reads one long word from the Altera memory space.
Input: Unsigned long integer (address offset)
Output: Unsigned long integer (data read)
Notes: As in the previous call the address offset value is passed into this call, but in this case the data value read from that address is returned.

IOCTL_ALT_GEN_RESET_RX_FIFOS
Function: Resets the Rx FIFOs.
Input: None
Output: None
Notes: Resets all eight receive FIFOs.

IOCTL_ALT_GEN_SET_RX_LEVEL
Function: Sets an Rx FIFO's almost full level.
Input: RX_LEVEL_LOAD structure
Output: None
Notes: The RX_LEVEL_LOAD structure has an eight-bit field: Channel – the number of the single receive FIFO to write to, and a 16-bit field Data – the almost full level to write. See the definition of RX_LEVEL_LOAD below.

```
typedef struct _RX_LEVEL_LOAD {
   UCHAR    Channel;
   USHORT   Data;
} RX_LEVEL_LOAD, *PRX_LEVEL_LOAD;
```

IOCTL_ALT_GEN_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.

IOCTL_ALT_GEN_ENABLE_INTERRUPT
Function: Enable the master interrupt.
Input: None
Output: None
Notes: This command must be run to allow the board to respond to interrupts. The master interrupt enable is disabled in the driver interrupt service routine. This command must then be run again to re-enable it.

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

IOCTL_ALT_GEN_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.

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