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

pci_alt

Linux Device Driver Documentation

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

pci_alt
Linux 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 C
Santa Cruz, CA 95060
(831) 457-8891
FAX: (831) 457-4793

Table of Contents

Introduction

The pci_alt driver is a Linux device driver for the PCI-Altera-485/LVDS from Dynamic Engineering. The PCI-Altera board has a (PLX) PCI-9054 and a Spartan-II Xilinx FPGA to implement the PCI interface for the board. In addition there is an Altera EP20K400EBC652 FPGA that is programmed over the PCI interface with a configuration file from the host file system. 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 bidirectional I/O channels.

When the PCI-Altera is recognized by the PCI bus configuration utility it will start the pci_alt driver, which requests the default Altera configuration file and programs the Altera FPGA. Configuration files are stored in the directory /usr/lib/hotplug/firmware/ and alt_atp.rbf is the name of the default configuration file. After the Altera is configured the design ID is read from the Altera base address and the appropriate Altera driver is loaded to control the Altera. Any additional Altera design files that will be loaded later should also be placed in the /usr/lib/hotplug/firmware/ directory.

The Altera is treated as a hot-swappable child of the pci_alt 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 based on the design ID read from the Altera base register (offset 0x2000 bits 31 - 24). If no dedicated Altera driver exists for the loaded configuration, the alt_gen driver will be loaded allowing for control of a generic Altera design. 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. The Control Calls specific a particular 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 drivers interact 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). The pci_alt driver was developed on Linux kernel version 2.6.35. If you are using a different version, some modification of the source code may be required.

Driver Installation

The source files and makefiles for the drivers and test applications are supplied in the driver archive file pci_altera.tar.bz2. Extract the directory structure to the computer where the driver is to be built.

From the top-level directory type “KBUILD_NOPEDANTIC=1 make” (necessary to suppress errors related to changing CFLAGS in Makefile). This will build the driver and test application object files. Next type “(sudo) make install” to copy the files to the target locations (/lib/modules/$(VERSION)/kernel/drivers/char/pci_alt/ for the drivers, /usr/local/bin/ for the test applications, /usr/lib/hotplug/firmware/ for the Altera configuration files, /sbin/hotplug/ for the firmware loader, and /etc/udev/rules.d/ for the configuration rule files) (you must be root for this to succeed). If so desired, type “make clean” to remove executable, object and interim files after installation.

After the driver is installed type (sudo) depmod to register the driver modules with the kernel (must be root).

Driver Startup

Install the hardware and boot the computer. Handles can be opened to a specific board by using the open() function call and passing in the appropriate device name.

Below is example code for opening a handle for device `dev_num`.

```
#typedef long HANDLE
#define       INPUT_SIZE 80

HANDLE   hpci_alt;

char  Name[INPUT_SIZE];
int   dev_num;

do
{
   printf("\nEnter target board number (starting with zero): \n");
   scanf("%d", &dev_num);
   if(dev_num < 0 || dev_num > NUM_PCI_ALT_DEVICES)
      printf("\nTarget board number %d out of range!\n", dev_num);
}
while(dev_num < 0 || dev_num > NUM_PCI_ALT_DEVICES);

sprintf(Name, "/dev/pci_alt_%d", dev_num);
hpci_alt = open(Name , O_RDWR);
if(hpci_alt < 2)
{
   printf("\n%s FAILED to open!\n", Name);
   return 1;
}
```

IO Controls

The driver uses ioctl() calls to configure the device and obtain status. The parameters passed to the ioctl() function include the handle obtained from the open() call, an integer command that is defined in the API header file and an optional parameter used to pass data in and/or out of the device. If both input and output parameters are needed, a union of the two is used for the I/O parameter. The ioctl commands defined for the pci_alt driver are listed below.

IOCTL_PCI_ALT_GET_INFO

Function: Returns the Driver Version, Xilinx Version, Switch value, Instance Number, and Transmit and Receive FIFO sizes.
Input: None
Output: PCI_ALT_DRIVER_DEVICE_INFO structure
Notes: The Xilinx version refers to the VHDL program used to program the Xilinx FPGA. The switch value is the configuration of the onboard dipswitch. This is selected by the User for whatever purpose needed (see the board silk screen for bit position and polarity). The Instance number should be zero for the first board seen and increments by one for each subsequent board. The FIFO sizes are dynamically detected when the driver starts up.

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 pci_alt_api.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 returned does not include FIFO or command queue reset bits, the Altera load bit, or the force interrupt bit as these are controlled by other calls. This command is used mainly for register testing.

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 pci_alt_api.h for information on interpreting this value.

IOCTL_PCI_ALT_LOAD_COMMAND

Function: Loads a write Tx / read Rx command into the command queue. These commands move data from the DMA FIFO to a Tx channel FIFO or from an Rx channel FIFO to the DMA FIFO.
Input: TRANSFER_COMMAND structure
Output: None
Notes: This command is used to specify Tx/Rx data routing and should be associated 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 available target FIFO space);
Transmit – true for a transmit command, false for a receive command.

IOCTL_PCI_ALT_GET_COMMAND_STATUS

Function: Returns 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 pci_alt_api.h for information on interpreting this value.

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: 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 pci_alt_api.h for information on determining this value.

IOCTL_PCI_ALT_GET_INT_CONFIG

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

IOCTL_PCI_ALT_GET_INT_STATUS

Function: Returns the interrupt status and clears 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 pci_alt_api.h for information on interpreting this value.

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.

IOCTL_PCI_ALT_GET_RX_DATA

Function: Reads an Rx data byte.
Input: Unsigned character
Output: Unsigned character
Notes: The channel 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 transferred.

IOCTL_PCI_ALT_PUT_DMA_DATA

Function: Loads 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: Reads 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: Resets the transmit and/or DMA FIFOs.
Input: FIFO_RESET structure
Output: None
Notes: The FIFO_RESET structure has two fields:
ResetDMA will cause the DMA FIFO to be reset if true.
TxResetMask is an eight-bit map of which Tx FIFOs to reset (1=> reset the corresponding FIFO, 0=> don't 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 bytes above empty that the programmable almost empty status will become true for the corresponding FIFO. See pci_alt_api.h for the definition of FIFO_LEVEL_LOAD.

IOCTL_PCI_ALT_GET_TX_LEVEL

Function: Returns the programmable almost empty level for a transmit FIFO.
Input: Unsigned character (FIFO number)
Output: Unsigned short integer
Notes: Returns the number of 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 bytes below full that the programmable almost full status will become true for the corresponding FIFO. See pci_alt_api.h for the definition of FIFO_LEVEL_LOAD.

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_WAIT_ON_INTERRUPT

Function: Inserts the calling process into a wait queue until an interrupt occurs.
Input: Time-out value in jiffies (unsigned long integer)
Output: None
Notes: This call is made in the user interrupt service routine to allow user-specified interrupt handlers for enabled interrupt conditions. The input parameter is a time-out value that causes the call to abort if the interrupt doesn't occur within the specified time. If the timeout is zero, the call will wait indefinitely. The DMA interrupts do not use this mechanism; they are controlled automatically by the driver.

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 user 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: Disables the master interrupt.
Input: None
Output: None
Notes: Used when user 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_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: This structure has two fields:
Status is the interrupt status value read in the ISR that occurred in response to the last user interrupt (an interrupt condition enabled by IOCTL_PCI_ALT_SET_INT_CONFIG). The interrupts that deal with the DMA transfers do not affect this value.
TimedOut is true if the specified timeout expired before an interrupt was seen.

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: a character array that specifies the name of the file to load. The file name cannot exceed 32 characters including the file extension and terminating null character.

Write

PCI-Altera transmit data is written to the device using the write command. A handle to the device, a pointer to a pre-allocated buffer that contains the data to write and an unsigned long integer that represents the number of data-bytes to write are passed to the write call. The driver will obtain physical addresses to the pages containing the data and will set-up a list of page descriptors in its list memory. The physical address of the first list entry is written to the board, which performs a bus-master scatter-gather DMA to transfer the data from the specified buffer to the DMA FIFO.

Read

PCI-Altera received data is read from the device using the read command. A handle to the device, a pointer to a pre-allocated buffer that will contain the data read and an unsigned long integer that represents the number of data-bytes to read are passed to the read call. The driver will obtain physical addresses to the buffer memory pages and will set-up a list of page descriptors in its list memory. The physical address of the first list entry is written to the board, which performs a bus-master scatter-gather DMA to transfer the data from the DMA FIFO to the specified buffer.

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