DYNAMIC ENGINEERING
435 Park Dr., Ben Lomond, Calif. 95005
831-336-8891 **Fax** 831-336-3840
http://www.dyneng.com
sales@dyneng.com
Est. 1988

PciLvds2R/T

Driver Documentation

Win32 Driver Model

Revision A
Corresponding Hardware
PCI-Lvds-2R: Revision C/D 10-2001-0204/0204
PCI-Lvds-2T: Revision C/D 10-2001-0903/0904

Lvds
WDM Device Driver for the
PCI-Lvds-2R & 2T
PCI based 2 channel Lvds receive
PCI based 2 channel Lvds transmit

Dynamic Engineering
435 Park Drive
Ben Lomond, CA 95005
831- 336-8891
831-336-3840 FAX

Manual Revision A. Revised July 13, 2006.

Table of Contents

Introduction

The Lvds driver is a Win32 driver model (WDM) device driver for the PCI-Lvds-2R and PCI-Lvds-2T from Dynamic Engineering. The PCI-Lvds-2R/2T boards have a PLX 9054 to implement the PCI interface and DMA data I/O for the board. In addition there is are 9 Xilinx FPGAs to handle the data I/O, and memory interface for two channels of Lvds parallel data.

When either of the PCI-Lvds-2R/2T are recognized by the PCI bus configuration utility, the Lvds driver will be started. Since both boards use the PLX-9054 to implement the PCI interface and therefore have the same vendor/device id numbers, a single driver must be used to service both boards. Once the driver is started it accesses the Timing register in the DMA Xilinx to distinguish between the two board types. A flag is set in the device object identifying the device as an 2R or 2T and one of two different GUIDs is registered to identify the device interface type. This allows the user to obtain handles to an explicit device by using the proper GUID and device number.

IO Control calls (IOCTLs) are used to configure and read status from the PCI-Lvds-2R/2T. Read and Write calls are used to move blocks of data in and out.

Note

This documentation will provide information about all calls made to the driver, and how the driver interacts with the devices for each of these calls.
For more detailed information on the hardware implementation, refer to the PCI-Lvds-2R and 2T user manuals (also referred to as the hardware manuals).

Driver Installation

There are several files provided in each driver package. These files include Lvds.sys, Lvds.inf, DDLv2r.h, DDLv2t.h, Lv2rGUID.h, Lv2tGUID.h, Lv2rDef.h, Lv2tDef.h, Lv2rTest.exe, Lv2tTest.exe, Lv2rTest source files, and Lv2tTest source files.

Windows 2000 Installation

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

With the PCI-Lvds-2R/2T hardware installed, power-on the PCI host computer and wait for the *Found New Hardware Wizard* dialogue window to appear.

- Select *Next*.
- Select *Search for a suitable driver for my device.*
- Select *Next*.
- Insert the disk prepared above in the desired drive.
- Select the appropriate drive e.g. *Floppy disk drives.*
- Select *Next*.
- The wizard should find the Lvds.inf file.
- Select *Next*.
- Select *Finish* to close the *Found New Hardware Wizard*.

Windows XP Installation

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

With the PCI-Lvds-2R/2T hardware installed, power-on the PCI host computer and wait for the *Found New Hardware Wizard* dialogue window to appear.

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

The DDLv2r.h and DDLv2t.h files are C header files that define the Application Program Interface (API) to the driver. The Lv2rGUID.h and Lv2tGUID.h files are C header files that define the device interface identifier for the PCI-Lvds-2R/2T. These files are required at compile time by any application that wishes to interface with the Lvds driver. The Lv2rDef.h and Lv2tDef.h files contain the relevant bit defines for the Lvds registers. These files are not needed for driver installation.

Lv2rTest.exe and Lv2rTest.exe file are sample Win32 console applications that makes calls into the Lvds driver to test the driver calls without actually

writing any application code. They are not required during the driver installation. Open a command prompt console window and type *Lv2rTest* or *Lv2tTest –d0 -?* to display a list of commands (the exe file must be in the directory that the window is referencing). The commands are all of the form *Lv2rTest –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.

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 globally unique identifiers (GUID), which are defined in Lv2rGUID.h and Lv2tGUID.h.

Below is example code for opening a handle for Lv2r device 0. To obtain a handle to an PCI-Lvds-2T, substitute GUID_DEVINTERFACE_LV2T for GUID_DEVINTERFACE_LV2R in two places.

```
// The maximum length of the device name for a given interface
#define MAX_DEVICE_NAME 256
// Device number to find
LONG                            devNum = 0;
// Handle to the device object
HANDLE                          hLv2r = 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_LV2R,
                                  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 devNum
if(!SetupDiEnumDeviceInterfaces(hDeviceInfo,
                                NULL,
                        (LPGUID)&GUID_DEVINTERFACE_LV2R,
                                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);
   }
}

// 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
hLv2r = CreateFile(deviceName,
                   GENERIC_READ    | GENERIC_WRITE,
                   FILE_SHARE_READ | FILE_SHARE_WRITE,
                   NULL,
                   OPEN_EXISTING,
                   NULL,
                   NULL);

if(hLv2r == 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_LV2R_GET_INFO

Function: Returns the Driver Version, Switch value, and Instance Number.
Input: none
Output: LV2R_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).

IOCTL_LV2R_GET_STATUS

Function: Returns the current status information.
Input: Channel number (UCHAR)
Output: LV2R_STATUS structure
Notes: See the bit and structure definitions in DDLv2r.h for information on interpreting this value.

IOCTL_LV2R_SET_MEMORY_CONFIG

Function: Sets the memory start and length for a receive channel.
Input: LV2R_MEMORY structure
Output: None
Notes: The start and length values refer to SDRAM addressing, which is measured in 64-bit words. This call redefines the current active channel for RAM fill and streaming I/O.

IOCTL_LV2R_GET_MEMORY_CONFIG

Function: Returns the memory start and length for a receive channel.
Input: Channel number (UCHAR)
Output: LV2R_MEMORY structure
Notes: The start and length values refer to SDRAM addressing, which is measured in 64-bit words.

IOCTL_LV2R_SET_CHANNEL_CONFIG

Function: Sets the channel configuration for a receive channel.
Input: LV2R_CHANNEL_CONFIG structure
Output: None
Notes: See the structure definition in DDLv2r.h for information on determining the input parameter. This call redefines the current active channel for RAM fill and streaming I/O.

IOCTL_LV2R_GET_CHANNEL_CONFIG

Function: Returns the channel configuration for a receive channel.
Input: Channel number (UCHAR)
Output: LV2R_CHANNEL_CONFIG structure
Notes: See the structure definition in DDLv2r.h for information on interpreting the output parameter.

IOCTL_LV2R_SET_TRIGGER_MODE

Function: Sets the trigger configuration for a receive channel.
Input: LV2R_TRIGGER_CONFIG structure
Output: None
Notes: See the structure and enum definitions in DDLv2r.h for information on determining the input parameter. This call redefines the current active channel for RAM fill and streaming I/O.

IOCTL_LV2R_GET_TRIGGER_MODE

Function: Returns the trigger configuration for a receive channel.
Input: Channel number (UCHAR)
Output: LV2R_TRIGGER_CONFIG structure
Notes: See the structure and enum definitions in DDLv2r.h for information on interpreting the output parameter.

IOCTL_LV2R_FILL_RAM

Function: Fills the memory space of the current active channel with a constant value.
Input: Fill pattern (ULONG)
Output: None
Notes: This call uses the current memory configuration for the current active channel to determine the memory space to fill.

IOCTL_LV2R_SELF_TEST

Function: Fills the memory space of a receive channel with an incrementing pattern.
Input: LV2R_SELF_TEST_CONFIG structure
Output: None
Notes: This call uses the current memory configuration for the specified channel to determine the memory space to fill. Unlike the previous call, the channel to fill is specified in the input data structure along with the starting 12-bit data pattern. Each subsequent data sample is incremented by one. Each 64-bit memory word contains four data samples.

IOCTL_LV2R_STREAM_POSITION

Function: Obtains streaming input buffer position.
Input: LV2R_STREAM_SET_INFO structure
Output: LV2R_STREAM_INFO structure
Notes: Gets the current streaming mode acquire data position. If the input is set to *Wait*, the call completes when the *SetPosition* point has been reached. Multiple calls to this function can be made for a single streaming acquire.

IOCTL_LV2R_SET_READ_OFFSET

Function: Sets the read position and RAM bank for the next ReadFile call.
Input: LV2R_READ_OFFSET structure
Output: None
Notes: Sets the offset from the start of a local RAM bank where the next ReadFile will begin. This call is valid for channels 0 and 4 only. If channel 0 is specified, RAM bank A is selected; if channel 4, RAM bank B.

IOCTL_LV2R_GET_READ_OFFSET

Function: Returns the read position and channel for the next ReadFile call.
Input: None
Output: LV2R_READ_OFFSET structure
Notes: Returns the offset from the start of a local RAM bank and the channel where the next ReadFile will operate. This call is valid for channels 0 and 4 only.

IOCTL_LV2R_SET_TIME_OUT

Function: Sets the timeout value for a ReadFile call.
Input: Timeout in milliseconds (ULONG)
Output: None
Notes: Sets the amount of time the driver should allow for an acquisition to complete. A value of zero causes the call to never timeout.

IOCTL_LV2R_GET_DATA_COUNTER

Function: Returns the number of data samples that have been stored during the current acquisition.
Input: Channel number (UCHAR)
Output: Cumulative stored sample counter value (ULONG)
Notes: The data sample counter is reset when a new acquisition is enabled.

IOCTL_LV2R_CHANNEL_STOP_ACQ

Function: Stops the data acquisition on the specified channel.
Input: Channel number (UCHAR)
Output: None

IOCTL_LV2R_CHANNEL_START_ACQ

Function: Starts data acquisition on a specified receive channel.
Input: Channel number (UCHAR)
Output: None

IOCTL_LV2R_RESET_ALL_INPUT_FIFOS

Function: Resets all input FIFOs.
Input: None
Output: None
Notes: The input FIFOs must be running off the PCI clock for the reset to be reliable. This call takes care of this requirement and then returns the configuration to its original value.

IOCTL_LV2R_IDENTIFY

Function: Flashes the user LED three times to identify the board.
Input: None
Output: None

IOCTL_LVDS_ISR_STATUS

Function: Returns the interrupt status read in the ISR from the last interrupt.
Input: None
Output: Value of interrupt status register (ULONG)
Notes: Returns the value of the DMA status register that was read in the previous ISR that was caused by a local interrupt. This is used to determine the interrupt cause for a local interrupt event. The latched status bits are also cleared by this call. This call is valid for both the PciLvds2R and 2T.

IOCTL_LVDS_REGISTER_EVENT

Function: Registers an Event object to be signaled when a local 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 local 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. This call is valid for both the PciLvds2R and 2T.

IOCTL_LVDS_ENABLE_INTERRUPT

Function: Sets the master interrupt enable for local interrupts and optionally sets a new done bit interrupt mask.
Input: Optional interrupt enable mask of the two done bits (UCHAR)
Output: None
Notes: This call enables local interrupts. If an input parameter is passed in, the interrupt enables in bits 16 to 23 of the DMA base control register are updated accordingly. The master interrupt enable is cleared in the driver ISR, so this call must be made to re-enable local interrupts. This call is valid for both the PciLvds2R and 2T.

IOCTL_LVDS_DISABLE_INTERRUPT

Function: Clears the master interrupt enable for local interrupts.
Input: None
Output: None
Notes: This call disables local interrupts when interrupt processing is no longer desired. This call is valid for both the PciLvds2R and 2T.

IOCTL_LVDS_FORCE_INTERRUPT

Function: Causes a local interrupt.
Input: None
Output: None
Notes: This call causes a local interrupt to occur. It is used to develop interrupt processing routines. This call is valid for both the PciLvds2R and 2T.

IOCTL_LV2T_GET_INFO

Function: Returns the Driver Version, Switch value, and Instance Number.
Input: none
Output: LV2T_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).

IOCTL_LV2T_GET_STATUS

Function: Returns the current status information.
Input: Channel number (UCHAR)
Output: LV2T_STATUS structure
Notes: See the bit and structure definitions in DDLv2t.h for information on interpreting this value.

IOCTL_LV2T_SET_MEMORY_CONFIG

Function: Sets the memory start, stop, loop addresses and configuration for a transmit channel.
Input: LV2T_MEMORY structure
Output: None
Notes: The values refer to SDRAM addressing, which is measured in 64-bit words. See the structure definition in DDLv2t.h for more information on interpreting this value.

IOCTL_LV2T_GET_MEMORY_CONFIG

Function: Returns the memory configuration for a transmit channel.
Input: Channel number (UCHAR)
Output: LV2T_MEMORY structure
Notes: See the structure definition in DDLv2t.h for information on interpreting the output parameter.

IOCTL_LV2T_SET_CHANNEL_CONFIG

Function: Sets the channel configuration for a transmit channel.
Input: LV2T_CHANNEL_CONFIG structure
Output: None
Notes: See the structure definition in DDLv2t.h for information on determining the input parameter.

IOCTL_LV2T_GET_CHANNEL_CONFIG

Function: Returns the channel configuration for a transmit channel.
Input: Channel number (UCHAR)
Output: LV2T_CHANNEL_CONFIG structure
Notes: See the structure definition in DDLv2t.h for information on interpreting the output parameter.

IOCTL_LV2T_SET_TIMING_CONFIG

Function: Sets the master timing configuration.
Input: LV2T_TIMING_CONFIG structure
Output: None
Notes: See the structure definition in DDLv2t.h for information on determining the input parameter.

IOCTL_LV2T_GET_TIMING_CONFIG

Function: Returns the master timing configuration.
Input: None
Output: LV2T_TIMING_CONFIG structure
Notes: See the structure definition in DDLv2t.h for information on determining the output parameter.

IOCTL_LV2T_SET_LOCAL_START

Function: Sets the local start configuration.
Input: LV2T_LOCAL_START_CONFIG structure
Output: None
Notes: See the structure definition in DDLv2t.h for information on determining the input parameter. Restart resets the Tx state machine to allow a new transmission and switches from the terminal idle pattern to the initial idle pattern. Start enables the local start (both local and master starts must be true to start a transmission).

IOCTL_LV2T_SET_MASTER_START

Function: Sets the master start configuration.
Input: LV2T_MASTER_START_CONFIG structure
Output: None
Notes: See the structure definition in DDLv2t.h for information on determining the input parameter. Enable enables the master start (both local and master starts must be true to start a transmission). Pulse generates a two clock wide output pulse to be used by other boards as an external trigger.

IOCTL_LV2T_START_READBACK_MODE

Function: Starts the Tx read-back mode allowing the RAM data to be read back over the PCI bus from a channels Tx Xilinx read-back port.
Input: Channel number (UCHAR)
Output: LV2T_RDBK_CONFIG structure
Notes: The structure saves the state of the start bit and the channel number. This structure can be passed into the stop read-back call to restore the original state, however, any data read while in read-back mode will be unavailable to be transmitted.

IOCTL_LV2T_STOP_READBACK_MODE

Function: Stops the Tx read-back mode.
Input: LV2T_RDBK_CONFIG structure
Output: None

IOCTL_LV2T_GET_TX_DATA

Function: Reads a block of 16 32-bit words from a channels Tx Xilinx read-back port.
Input: Channel number (UCHAR)
Output: LV2T_RDBK_DATA structure (16 long words)
Notes: Any data read while in read-back mode will be unavailable to be transmitted when read-back mode is terminated.

IOCTL_LV2T_SET_WRITE_OFFSET

Function: Stops the Tx read-back mode.
Input: LV2T_WRITE_OFFSET structure
Output: None
Notes: Sets the offset from the start of a local RAM bank where the next WriteFile will begin. This call is valid for channels 0 and 4 only. If channel 0 is specified, RAM bank A is selected; if channel 4, RAM bank B.

IOCTL_LV2T_GET_WRITE_OFFSET

Function: Returns the write position and channel for the next WriteFile call.
Input: None
Output: LV2T_WRITE_OFFSET structure
Notes: Returns the offset from the start of a local RAM bank and the channel where the next WriteFile will operate. This call is valid for channels 0 and 4 only.

IOCTL_LV2T_RESET_ALL_OUTPUT_FIFOS

Function: Resets all output FIFOs.
Input: None
Output: None
Notes: The output FIFOs must be running off the PCI clock for the reset to be reliable. This call takes care of this requirement and then returns the configuration to its original value.

IOCTL_LV2T_IDENTIFY

Function: Flashes the user LED three times to identify the board.
Input: None
Output: None

Write

PCI-Lvds-2T 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 int that represents the requested size of the transfer in bytes, a pointer to an unsigned long int to contain the number of bytes actually written, and a pointer to an optional Overlapped structure for performing asynchronous IO.

Read

PCI-Lvds-2R 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 int that represents the size of the requested transfer in bytes, a pointer to an unsigned long int 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
435 Park Dr.
Ben Lomond, CA 95005
831-336-8891
831-336-3840 fax
support@dyneng.com