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

PCIeBiSerialDb37 LM9 Base & Channel

Driver Documentation

Win32 Driver Model

Manual Revision A
Corresponding Hardware: Revision A
10-2009-0401
Corresponding Firmware:
LM9: Design 1, Revision 1

LM9Base & LM9Chan
WDM Device Drivers for the
PcieBiserialDb37Lm9

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

Manual Revision A Revised Oct. 12, 2009

This document contains information of proprietary interest to Dynamic Engineering. It has been supplied in confidence and the recipient, by accepting this material, agrees that the subject matter will not be copied or reproduced, in whole or in part, nor its contents revealed in any manner or to any person except to meet the purpose for which it was delivered.

Dynamic Engineering has made every effort to ensure that this manual is accurate and complete. Still, the company reserves the right to make improvements or changes in the product described in this document at any time and without notice. Furthermore, Dynamic Engineering assumes no liability arising out of the application or use of the device described herein.

The electronic equipment described herein generates, uses, and can radiate radio frequency energy. Operation of this equipment in a residential area is likely to cause radio interference, in which case the user, at his own expense, will be required to take whatever measures may be required to correct the interference.

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.

Connection of incompatible hardware is likely to cause serious damage.

Table of Contents

Introduction

The LM9Base and LM9Chan drivers are Win32 driver model (WDM) device drivers for the PCIeBiSerialDb37Lm9 from Dynamic Engineering.

The LM9 driver package has two parts. The driver is installed into the Windows® OS, and the User Application "Userap" executable.

The driver is delivered as installed or executable items to be used directly or indirectly by the user. The Userap code is delivered in source form [C] and is for the purpose of providing a reference to using the driver.

UserAp is a stand-alone code set with a simple, and powerful menu plus a series of "tests" that can be run on the installed hardware. Each of the tests execute calls to the driver, pass parameters and structures, and get results back. With the sequence of calls demonstrated, the functions of the hardware are utilized for loop-back testing. The software is used for manufacturing test at Dynamic Engineering. For example most Dynamic Engineering PCI based designs support DMA. DMA is demonstrated with the memory based loop-back tests. The tests can be ported and modified to fit your requirements.

The test software can be ported to your application to provide a running start. It is recommended to port the switch and status tests to your application to get started. The tests are simple and will quickly demonstrate the end-to-end operation of your application making calls to the driver and interacting with the hardware.

The menu allows the user to add tests, to run sequences of tests, to run until a failure occurs and stop or to continue, to program a set number of loops to execute and more. The user can add tests to the provided test suite to try out application ideas before committing to your system configuration. In many cases the test configuration will allow faster debugging in a more controlled environment before integrating with the rest of the system.

The hardware has features common to the board level and features that are set apart in "channels". The channels have the same offsets within the channel, and the same status and control bit locations allowing for symmetrical software in the calling routines. The driver supports the channels with a variable passed in to identify which channel is being accessed. The hardware manual defines the pinout for each channel and the bitmaps and detailed configurations for each channel. The driver handles all aspects of interacting with the channels and base features.

We strive to make a useable product, and while we can guarantee operation we can't foresee all concepts for client implementation. If you have suggestions for extended features, special calls for particular set-ups or whatever please share them with us, [engineering@dyneng.com] and we will consider and in many cases add them.

The PCIeBiSerialDb37LM9 design has a Spartan3 Xilinx FPGA to implement the PCI interface, FIFO's and protocol control and status for the IO. The IO are grouped into two ports; both part of channel 0. A Transmit port which sends data to the ARC-210 device and a Receiver port are provided. Please refer to the HW manual for a much more complete description of the HW features.

When the PCIeBiSerialDb37Lm9 board is recognized by the PCI bus configuration utility it will start the LM9Base driver which will create a device object for each board, initialize the hardware, create a child devices for the channel and request loading of the LM9Chan driver. The LM9Chan driver will create a device object for the I/O channel and perform initialization on the channel. IO Control calls (IOCTLs) are used to configure the board and read status. Read and Write calls are used to move blocks of data in and out of the device.

Note

This documentation will provide information about all calls made to the drivers, and how the drivers interact with the device for each of these calls. For more detailed information on the hardware implementation, refer to the PCIeBiSerialDb37Lm9 user manual (also referred to as the hardware manual).

Driver Installation

There are several files provided in each driver package. These files include driver: LM9Base.sys, PcieBisDb37LM9.inf, DDLM9Base.h, LM9BaseGUID.h, LM9Chan.sys, DDLM9Chan.h, LM9ChanGUID.h.   Userap: User Application source files.

LM9BaseGUID.h and LM9ChanGUID.h are C header files that define the device interface identifiers for the drivers. DDLM9Base.h and DDLM9Chan.h files 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. The files are included with the Userap fileset.

Windows 2000 Installation

Copy PcieBisDb37LM9.inf, LM9Base.sys and LM9Chan.sys to a floppy disk, or CD if preferred. In some cases the files can be accessed over a network or from local HDD. Substitute the network address for the floppy instructions to proceed with an over the network installation.

With the 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 PmcLM9.inf file.
_ Select ***Next***.
_ Select ***Finish*** to close the ***Found New Hardware Wizard***.
The system should now see the channels and reopen the ***New Hardware Wizard.***
Repeat this for each channel as necessary.

Windows XP Installation

Copy PcieBisDb37LM9.inf, LM9Base.sys and LM9Chan.sys to a floppy disk, or CD if preferred. In some cases the files can be accessed over a network or from local HDD. Substitute the network address for the floppy instructions to proceed with an over the network installation.

With the 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 system should now see the channels and reopen the ***New Hardware Wizard.***
Proceed as above for each channel as necessary.

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 (GUIDs), which are defined in LM9BaseGUID.h and LM9ChanGUID.h.

The User Application software contains a file called “main.c”. Main has the initialization needed to get the handles to the base and channel assets of the installed PCIeBiSerialDb37Lm9 device.

The main file provided is designed to work with our test menu and includes user interaction steps to allow the user to select which board is being tested in a multiple board environment. The integrator can hardcode for single board systems or use an automatic loop to operate in multiple board systems without using user interaction. For multiple user systems it is suggested that the board number is associated with a switch setting so the calls can be associated with a particular board from a physical point of view.

IO Controls

The drivers use IO Control calls (IOCTLs) to configure the device. 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 LM9Base driver are described below:
Please note that the address map is included in the DD file for reference when writing your own driver for a different OS.

IOCTL_LM9_BASE_GET_INFO

Function: Return the Instance Number, Switch value, PLL device ID, Xilinx rev and Current Driver Version
Input: None
Output: LM9_BASE_DRIVER_DEVICE_INFO : Structure
Notes: Switch value is the configuration of the on-board dip-switch that has been set by the User (see the board silk screen for bit position and polarity). The PLL ID is the device address of the PLL device. This value, which is set at the factory, is usually 0x69 but may also be 0x6A. See DDLM9Base.h for the definition of LM9_BASE_DRIVER_DEVICE_INFO.

IOCTL_LM9_BASE_LOAD_PLL_DATA

Function: Loads the internal registers of the PLL.
Input: LM9_BASE_PLL_DATA structure
Output: None
Notes:

IOCTL_LM9_BASE_READ_PLL_DATA
Function: Returns the contents of the PLL's internal registers
Input: None
Output: LM9_BASE_PLL_DATA structure
Notes: The register data is output in the LM9_BASE_PLL_DATA structure in an array of 40 bytes.

IOCTL_LM9_BASE_SET_BASEREG
Function: Write to Base Control Register - general access to base control register of card, use with bit definitions
Input: ULONG
Output: none
Notes: Use for general purpose – bit mapped access to the base control register.

IOCTL_LM9_BASE_GET_BASEREG
Function: Read from Base Control Register - general access from base control register of card, use with bit definitions
Input: none
Output: ULONG
Notes: Use for general purpose – bit mapped access to the base control register.

IOCTL_LM9_BASE_GET_STATUS
Function: Read from Status Register
Input: none
Output: ULONG
Notes: Use for general purpose – bit mapped access from the register. See DDLM9Base.h for bit map information. See the HW manual for exact definitions of bits.

// GPIO Control Section
//   IO not currently used by the ARC-210 IF is available for GP use
//   Bits are aligned to “0” in registers and remapped to actual IO
//   6 used for ARC-210, 12 available in GPIO

IOCTL_LM9_BASE_SET_GPIO_TERM
Function: Write to GPIO Termination Control Register
Input: ULONG
Output: none
Notes: Set bits to turn on termination for those bits

IOCTL_LM9_BASE_GET_GPIO_TERM
Function: Read from GPIO Termination Control Register
Input: none
Output: ULONG
Notes:

IOCTL_LM9_BASE_SET_GPIO_DIR
Function: Write to GPIO Direction Control Register
Input: ULONG
Output: none
Notes: Set bits to select transmit, clear for receive

IOCTL_LM9_BASE_GET_GPIO_DIR
Function: Read from GPIO Direction Control Register
Input: none
Output: ULONG
Notes:

IOCTL_LM9_BASE_SET_GPIO_DATA
Function: Write to GPIO Data Control Register
Input: ULONG
Output: none
Notes: Set output data pattern here. Only TX enabled bits will be transmitted

IOCTL_LM9_BASE_GET_GPIO_DATA
Function: Read from GPIO Data Control Register
Input: none
Output: ULONG
Notes: Read back of control register. For IO Data see next IOCTl

IOCTL_LM9_BASE_GET_GPIO
Function: Read from GPIO IO lines
Input: none
Output: ULONG
Notes: Read all lines whether TX or RX defined. Use previous IOCTL for read-back of Data register.

The IOCTLs defined for the LM9Chan driver are described below:
In the LM9 implementation both the Transmitter and the Receiver interface are implemented within the same channel (0). The Receiver accepts data from the external equipment. The Transmitter provides data to the external equipment.

Address and bit map information is included in the DDLM9Chan.h file to support those who are writing drivers for other OS.

IOCTL_LM9_CHAN_GET_INFO
Function: Return the Instance Number and Current Driver Version
Input: None
Output: LM9_CHAN_DRIVER_DEVICE_INFO structure
Notes: See the definition of LM9_CHAN_DRIVER_DEVICE_INFO in the DDLM9Chan.h header file.

IOCTL_LM9_CHAN_GET_STATUS
Function: Return the value of the status register and clear latched bits
Input: None
Output: Status register value(ULONG)
Notes: Latched interrupt and error status are cleared by write-back. See quick reference status bits below. Defines available in DDLM9Chan.h Detailed definitions are available in the HW manual.

```
#define STAT_TX_FIFO_MT           0x00000001 //0 set when TX FIFO is empty
#define STAT_TX_FIFO_AE           0x00000002 //1 set when TX FIFO is Almost
                                  Empty
#define STAT_TX_FIFO_FULL         0x00000004 //2 set when TX FIFO is Full

#define STAT_RX_FIFO_MT           0x00000010 //4 set when RX FIFO is Empty
#define STAT_RX_FIFO_AF           0x00000020 //5 set when RX FIFO is Almost
                                  Full
#define STAT_RX_FIFO_FULL         0x00000040 //6 set when RX FIFO is Full

#define STAT_RX_PARITY_ERROR      0x00000200 //9 Set when RX state machine
                                  creates an interrupt, latched - clear with write
#define STAT_TX_AE_INT_LAT        0x00000400 //10 Transmit FIFO Interrupt
                                  occurred, latched - clear with write
#define STAT_RX_AF_INT_LAT        0x00000800 //11 Receive FIFO Interrupt
                                  occurred, latched - clear with write

#define STAT_WR_DMA_ERR           0x00001000 //12 write DMA error, latched -
                                  clear with write
#define STAT_RD_DMA_ERR           0x00002000 //13 read DMA error, latched -
```

```
                                  clear with write
#define STAT_WR_DMA_INT           0x00004000 //14 write DMA Interrupt, latched -
                                  clear with write
#define STAT_RD_DMA_INT           0x00008000 //15 read DMA Interrupt, latched -
                                  clear with write

#define STAT_TXPKTDONE            0x00010000 //16 set when TX packet
                                  completed, latched - clear with write
#define STAT_RXPKTDONE            0x00020000 //17 set when RX packet
                                  completed, latched - clear with write
#define STAT_RX_OVFL_ERR          0x00040000 //18 Set when RX overflow error
                                  occurred, latched - clear with write
#define STAT_TX_UNFL_ERR          0x00080000 //19 Set when TX underflow error
                                  occurred, latched - clear with write

#define STAT_RX_IDLE              0x00100000 //20 set when RX is in Idle state
#define STAT_TX_IDLE              0x00200000 //21 set when TX is in Idle state
#define STAT_DMA_RD_IDLE          0x00400000 //22 set when Burst Out [read]
                                  DMA state-machine is in the idle state
#define STAT_DMA_WR_IDLE          0x00800000 //23 set when Burst In [write] DMA
                                  state-machine is in the idle state

#define TX_PACKET_FIFO_MT         0x01000000 //24 Tx Packet FIFO is MT when
                                  set
#define TX_PACKET_FIFO_FULL       0x02000000 //25 Tx Packet FIFO is FULL
                                  when set
#define RX_PACKET_FIFO_MT         0x04000000 //26 Rx Packet FIFO is MT when
                                  set
#define RX_PACKET_FIFO_FULL       0x08000000 //27 Rx Packet FIFO is FULL
                                  when set

#define LOCAL_INT                 0x40000000 //30 non DMA interrupt status
                                  before channel mask
#define STAT_ACTIVE_INT           0x80000000 //31 channel interrupt is active
                                  [after mask and includes DMA]
```

IOCTL_LM9_CHAN_CLR_STATUS
Function: Clear Error Bits latched and not cleared by status read
Input: ULONG
Output: none
Notes: Clear latched error bits. Allows polling on FIFO status without losing potential Error conditions. Write back with same bit position set to clear. Defines available in DDLM9Chan.h Detailed definitions are available in the HW manual.

IOCTL_LM9_CHAN_SET_FIFO_LEVELS
Function: Sets the transmitter almost empty and receiver almost full levels for the channel.
Input: LM9_CHAN_FIFO_LEVELS structure
Output: None
Notes: The FIFO counts are compared to these levels to determine the value of the STAT_TX_FF_AMT and STAT_RX_FF_AFL status bits.

IOCTL_LM9_CHAN_GET_FIFO_LEVELS
Function: Returns the transmitter almost empty and receiver almost full levels for the channel.
Input: None
Output: LM9_CHAN_FIFO_LEVELS structure
Notes:

IOCTL_LM9_CHAN_GET_FIFO_COUNTS
Function: Returns the number of data words in FIFO's.
Input: None
Output: LM9_CHAN_FIFO_COUNTS structure
Notes: Returns the actual TX FIFO data counts and count including DMA pipeline RX FIFO.

IOCTL_LM9_CHAN_RESET_FIFOS
Function: Resets one or both internal FIFOs for the referenced channel.
Input: LM9_FIFO_SEL enumeration type See structure definition in DDLM9Chan.h
Output: None
Notes: Resets Transmit, Receive, Both (Transmit and Receive) .

IOCTL_LM9_CHAN_REGISTER_EVENT

Function: Registers an event to be signaled when an interrupt occurs.
Input: Handle to the 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_LM9_CHAN_ENABLE_INTERRUPT

Function: Enables the channel 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 when a user interrupt is serviced. Therefore this command must be run after each interrupt occurs to re-enable it.

IOCTL_LM9_CHAN_DISABLE_INTERRUPT

Function: Disables the channel Master Interrupt.
Input: None
Output: None
Notes: This call is used when user interrupt processing is no longer desired.

IOCTL_LM9_CHAN_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 channel master interrupt is enabled. This IOCTL is used for development, to test interrupt processing. Board level master interrupt also needs to be set.

IOCTL_LM9_CHAN_GET_ISR_STATUS

Function: Returns the interrupt status read in the ISR from the last user interrupt.
Input: None
Output: Interrupt status value (unsigned long integer)
Notes: Returns the interrupt status that was read in the interrupt service routine of the last interrupt caused by one of the enabled channel interrupts. The interrupts that deal with the DMA transfers do not affect this value. Masked version of channel status.

IOCTL_LM9_CHAN_SWW_TX_FIFO

Function: Writes a 32-bit data word to the transmit FIFO.
Input: FIFO word (unsigned long integer)
Output: none
Notes: Used to make single-word accesses to the transmit FIFO instead of using DMA.

IOCTL_LM9_CHAN_SWR_RX_FIFO

Function: Returns a 32-bit data word from the receive FIFO.
Input: None
Output: FIFO word (unsigned long integer)
Notes: Used to make single-word accesses to the receive FIFO instead of using DMA. Please note, Data read from this port is no longer available in the FIFO for DMA or other use.

IOCTL_LM9_CHAN_SET_CONT

Function: write to Channel Control register using structure
Input: LM9_CHAN_CONT
Output: None
Notes: See DDLM9Chan.h for structure. See below for quick reference.

IOCTL_LM9_CHAN_GET_CONT

Function: Read from Channel Control register using structure
Input: None
Output: LM9_CHAN_CONT
Notes: See DDLM9Chan.h for structure. See below for quick reference.

```
FifoTestEn;   // BiPass Mode Control
MIntEn;       // Master Interrupt Enable
WrDmaEn;      // Write DMA Interrupt Enable
RdDmaEn;      // Read DMA Interrupt Enable
TxUrgent;   // Set for higher priority TX DMA processing
RxUrgent;   // Set for higher priority RX DMA processing
```

IOCTL_LM9_CHAN_SET_TX
Function: write to Channel Tx Control register using structure
Input: LM9_CHAN_TX_CONTROL
Output: None
Notes: See DDLM9Chan.h for structure.

IOCTL_LM9_CHAN_GET_TX
Function: Read from Channel Master Control register using structure
Input: None
Output: LM9_CHAN_TX_CONTROL
Notes: See DDLM9Chan.h for structure.

Quick Reference:

BOOLEAN	TxStart;	//0 start TX state machine
BOOLEAN	TxIntEn;	//2 set to enable TX interrupt
BOOLEAN	TxAEIntEn;	//3 set to enable TX FIFO based interrupt [almost empty]
BOOLEAN	TxUnFlEn;	//4 set to enable UnderFlow interrupt
BOOLEAN	TxByteOrder;	//5 set to reverse bytes before sending
BOOLEAN	TxBitOrder;	//6 set to reverse bits before sending
BOOLEAN	TxClkPol;	//7 Set to change on falling edge [rising valid] clear to change on rising edge [falling valid]
BOOLEAN	TxRegPacket;	//8 Set to use register data path instead of FIFO path
BOOLEAN	TxParity;	//9 Set to use odd parity else use even parity
BOOLEAN	TxClockDir;	//12 Set to enable SENDTIMING to be transmitted instead of received
BOOLEAN	TxClockSrc;	//13 Set to use divided PLL else use PLL rate
BOOLEAN	TxStartBit;	//14 Start bit sense - should be opposite of Marking state
BOOLEAN	TxMarkBit;	//15 Marking bit sense - should be opposite of Start

IOCTL_LM9_CHAN_SET_TX_COUNT
Function: write to Channel TXCount register
Input: ULONG
Output: None
Notes: Set the count for the Transmit packet count in bytes. Please note that the control bit “TxRegPacket” selects whether this register or the Tx Packet FIFO is used as the source of the defined packets.

IOCTL_LM9_CHAN_GET_TX_COUNT
Function: Read from Channel TX Count Register
Input: None
Output: ULONG
Notes:

IOCTL_LM9_CHAN_TX_PACKET_FIFO_WRITE
Function: write to Channel TX Packet FIFO
Input: ULONG
Output: None
Notes: Set the count for the Transmit packet count in bytes. Please note that the control bit “TxRegPacket” selects whether this register or the Tx Packet FIFO is used as the source of the defined packets. FIFO is 2K x 32. Status available for Full and Empty conditions in Status register.

IOCTL_LM9_CHAN_TX_PACKET_FIFO_READ
Function: Read from Channel TX Packet FIFO
Input: None
Output: ULONG
Notes: Read back port for test purposes. Once read, data is no longer in the FIFO for transmission purposes.

IOCTL_LM9_CHAN_SET_RX
Function: write to Channel Receiver Control register using structure
Input: LM9_CHAN_RX_CONTROL
Output: None
Notes: See DDLM9Chan.h for structure.

IOCTL_LM9_CHAN_GET_RX
Function: Read from Channel Receiver Control register using structure
Input: None
Output: LM9_CHAN_RX_CONTROL
Notes: See DDLM9Chan.h for structure.

Quick Reference:

BOOLEAN	RxStart;	//0 set to begin RX Data Acquisition
BOOLEAN	RxParityErrEn;	//1 set to enable Parity Error Interrupt
BOOLEAN	RxIntEn;	//2 set to enable RX interrupt
BOOLEAN	RxAFIntEn;	//3 set to enable RX FIFO based interrupt [almost full]
BOOLEAN	RxOvFlEn;	//4 set to enable RX OverFlow interrupt
BOOLEAN	RxByteOrder;	//5 set to reverse bytes after receiving
BOOLEAN	RxBitOrder;	//6 set to reverse bits before sending
BOOLEAN	RxClkPol;	//7 Set to use rising edge of clock or clear for falling edge valid data
BOOLEAN	RxParity;	//9 Set to use odd parity else use even parity
BOOLEAN	RxTimeOutEn;	//10 Set to use timeout control, 0 to ignore
BOOLEAN	RxStartBit;	//14 Start bit sense - should be opposite of Marking state
BOOLEAN	RxMartBit;	//15 Marking bit sense - should be opposite of Start

IOCTL_LM9_CHAN_SET_RX_COUNT
Function: write to Channel Receiver Count register
Input: ULONG
Output: None
Notes: Set the count for the size of a data block to be received. The count is in Bytes. If not known the timeout feature can be used.

IOCTL_LM9_CHAN_GET_RX_COUNT
Function: Read from Channel Receiver Count register
Input: None
Output: ULONG
Notes:

IOCTL_LM9_CHAN_RX_PACKET_FIFO_READ
Function: Read from Channel RX Packet FIFO
Input: None
Output: ULONG
Notes: FIFO is 2K x 32. Status available for Full and Empty conditions in Status register. Packet definitions are size of data stored in Data FIFO. Status should be used to validate Packet FIFO. If “over read” data will be last data. Can be read in response to RX Packet Interrupt and then corresponding data read from Data FIFO.

IOCTL_LM9_CHAN_RX_SET_TIMEOUT
Function: write to Channel Receiver TimeOut Register
Input: ULONG
Output: None
Notes: Set the Time Out length based on 33 MHz clock. [Program the number of periods of the reference clock desired.] When a gap between bytes is greater than the Time Out as defined in this register the previously captured data is “packetized” by storing the Packet Size and setting the RX Packet Completed bit. Additional data will become part of the next Packet received.

IOCTL_LM9_CHAN_RX_GET_TIMEOUT
Function: Read from Channel Receiver TimeOut Register
Input: None
Output: ULONG
Notes:

Write

DMA data is written to the referenced I/O channel device using the write command. Writes are executed using the Win32 function WriteFile() and passing in the handle to the I/O channel 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

DMA data is read from the referenced I/O channel device using the read command. Reads are executed using the Win32 function ReadFile() and passing in the handle to the I/O channel 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.

Examples of using DMA are provided in the reference software FIFO and IO loop-tests.

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 Street, Suite C
Santa Cruz, CA 95060
831-457-8891
831-457-4793 Fax

support@dyneng.com

Appendix

Reference copy of structures for evaluation

The following structures shown are available in the DDORBChan.h and DDLM9Base.h files included with the driver. The structures are included here for your evaluation when considering the driver package. The electronic versions included with the driver should be used with your project. The names track the register bit definitions. For details about particular signals please refer to the HW manual.

Base:

```
#define PLL_MESSAGE1_SIZE       16
#define PLL_MESSAGE2_SIZE       24
#define PLL_MESSAGE_SIZE        (PLL_MESSAGE1_SIZE + PLL_MESSAGE2_SIZE)

 // Driver/Device information
typedef struct _LM9_BASE_DRIVER_DEVICE_INFO
{
  UCHAR    DriverVersion;
  UCHAR    XilinxVersion;
  UCHAR    XilinxDesign;
  UCHAR    PllDeviceId;
  UCHAR    SwitchValue;
  ULONG    InstanceNumber;
} LM9_BASE_DRIVER_DEVICE_INFO, *PLM9_BASE_DRIVER_DEVICE_INFO;

typedef struct _LM9_BASE_PLL_DATA
{
  UCHAR    Data[PLL_MESSAGE_SIZE];
} LM9_BASE_PLL_DATA, *PLM9_BASE_PLL_DATA;
```

Channel:

```
typedef struct _LM9_CHAN_DRIVER_DEVICE_INFO
{
  UCHAR    DriverVersion;
  ULONG    InstanceNumber;
} LM9_CHAN_DRIVER_DEVICE_INFO, *PLM9_CHAN_DRIVER_DEVICE_INFO;

typedef enum _LM9_CHAN_FIFO_SEL {LM9_MAS, LM9_TAR, LM9_BOTH}
LM9_CHAN_FIFO_SEL, *PLM9_CHAN_FIFO_SEL;

typedef struct _LM9_CHAN_FIFO_LEVELS
{
  USHORT   AlmostFull;                // Set to control Master HW with Almost full definition
  USHORT   AlmostEmpty;               // set to control Target HW with Almost Empty definition,
Also controls Interrupt request
} LM9_CHAN_FIFO_LEVELS, *PLM9_CHAN_FIFO_LEVELS;

typedef struct _LM9_CHAN_FIFO_COUNTS
{
  USHORT   RxCountwPipe;
  USHORT   TxCount;
} LM9_CHAN_FIFO_COUNTS, *PLM9_CHAN_FIFO_COUNTS;

typedef struct _LM9_CHAN_CONT
{
  BOOLEAN                             FifoTestEn;// BiPass Mode Control
  BOOLEAN                             MIntEn;    // Master Interrupt Enable
  BOOLEAN                             WrDmaEn;   // Write DMA Interrupt Enable
  BOOLEAN                             RdDmaEn;   // Read DMA Interrupt Enable
  BOOLEAN                             TxUrgent;  // Set for higher priority TX DMA processing
  BOOLEAN                             RxUrgent;  // Set for higher priority RX DMA processing
} LM9_CHAN_CONT, *PLM9_CHAN_CONT;
```

```
typedef struct _LM9_CHAN_RX_CONTROL
{
   BOOLEAN             RxStart;          //0 set to begin RX Data Acquisition
   BOOLEAN             RxParityErrEn;    //1 set to enable Parity Error Interrupt
   BOOLEAN             RxIntEn;          //2 set to enable RX interrupt
   BOOLEAN             RxAFIntEn;        //3 set to enable RX FIFO based interrupt [almost full]
   BOOLEAN             RxOvFlEn;         //4 set to enable RX OverFlow interrupt
   BOOLEAN             RxByteOrder;      //5 set to reverse bytes after receiving
   BOOLEAN             RxBitOrder;       //6 set to reverse bits before sending
   BOOLEAN             RxClkPol;         //7 Set to use rising edge of clock or clear for falling edge valid
   BOOLEAN             RxParity;         //9 Set to use odd parity else use even parity
   BOOLEAN             RxTimeOutEn;      //10 Set to use timeout control, 0 to ignore
   BOOLEAN             RxStartBit;       //14 Start bit sense - should be opposite of Marking state
   BOOLEAN             RxMartBit;        //15 Marking bit sense - should be opposite of Start bit
} LM9_CHAN_RX_CONTROL, *PLM9_CHAN_RX_CONTROL;
```

```
typedef struct _LM9_CHAN_TX_CONTROL
{
   BOOLEAN             TxStart;          //0 start TX state machine
   BOOLEAN             TxIntEn;          //2 set to enable TX interrupt
   BOOLEAN             TxAEIntEn;        //3 set to enable TX FIFO based interrupt [almost empty]
   BOOLEAN             TxUnFlEn;         //4 set to enable UnderFlow interrupt
   BOOLEAN             TxByteOrder;      //5 set to reverse bytes before sending
   BOOLEAN             TxBitOrder;       //6 set to reverse bits before sending
   BOOLEAN             TxClkPol;         //7 Set to change on falling edge clear to change on rising edge
   BOOLEAN             TxRegPacket;      //8 Set to use register data path instead of FIFO path
   BOOLEAN             TxParity;         //9 Set to use odd parity else use even parity
   BOOLEAN             TxClockDir;       //12 Set to enable SENDTIMING to be transmitted
   BOOLEAN             TxClockSrc;       //13 Set to use divided PLL else use PLL rate
   BOOLEAN             TxStartBit;       //14 Start bit sense - should be opposite of Marking state
   BOOLEAN             TxMarkBit;        //15 Marking bit sense - should be opposite of Start bit
} LM9_CHAN_TX_CONTROL, *PLM9_CHAN_TX_CONTROL;
```