**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

# PCIe-Biserial-DB37-LM9

## ARC-210 Interface
## Windows 10 WDF Driver Documentation

### Developed with Windows Driver Foundation
### Ver1.19

Revision 01
Corresponding Hardware: Revision 04

**PMC-Biserial-DB37-LM9**
WDF Device Drivers for the
PMC-BiS Db37 Lm9

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


Manual Revision A. Revised November 26, 2019.

# Table of Contents

# Introduction

The PCIeBiSerialDb37Lm9 driver was developed with the Windows Driver Foundation version 1.19 (WDF) from Microsoft, specifically the Kernel-Mode Driver Framework (KMDF).

The PCIeBiSerialDb37Lm9 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, FIFOs 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 Lm9BasePublic.h, Lm9ChanPublic.h, Lm9Base.inf, Lm9Chan.inf, lm9base.cat, lm9chan.cat, Lm9Base.sys, and Lm9Chan.sys.

Lm9BasePublic.h and Lm9ChanPublic.h are the C header file that defines the Application Program Interface (API) for the PCIeBiSDb37Lm9 driver. This file is required at compile time by any application that wishes to interface with the drivers, but is not needed for driver installation.

### Windows 10 Installation

Copy Lm9Base.inf, Lm9Chan.inf, lm9base.cat, lm9chan.cat, Lm9Base.sys, and Lm9Chan.sys (Win10 version) to a CD or USB memory device as preferred.

With the PCIe BiSerial DB37 LM9 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***.
- Select ***Let me pick from a list of device drivers on my computer***.
- Select ***Next***.
- Select ***Have Disk*** and enter the path to the device prepared above.
- Select ***Next***.
- Select ***Close*** to close the update window.

The system should now display the PcieBiSDb37Lm9 PCI adapter in the Device Manager.

***** 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 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 Lm9BasePublic.h and Lm9ChanPublic.h. See main.c in the PcieBiSDb37Lm9UserApp project for an example of how to acquire a handle to the 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(), 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:**

### IOCTL_LM9_BASE_GET_INFO

***Function:*** Returns the device driver version, Xilinx flash revision, PLL device ID, user switch value, and device instance number.
***Input:*** None
***Output:*** LM9_BASE_DRIVER_DEVICE_INFO structure
***Notes:*** The switch value is the configuration of the 8-bit 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 device number. See the definition of LM9_BASE_DRIVER_DEVICE_INFO below.

```
// 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;
```

### IOCTL_LM9_BASE_LOAD_PLL_DATA

***Function:*** Writes to the internal registers of the PLL.
***Input:*** LM9_BASE_PLL_DATA structure
***Output:*** None
***Notes:*** The LM9_BASE_PLL_DATA structure has only one field: Data – an array of 40 bytes containing the PLL register data to write. See below for the definition of LM9_BASE_PLL_DATA.

```
 // Structures for IOCTLs
#define PLL_MESSAGE1_SIZE        16
#define PLL_MESSAGE2_SIZE        24
#define PLL_MESSAGE_SIZE        (PLL_MESSAGE1_SIZE + PLL_MESSAGE2_SIZE)

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

### IOCTL_LM9_BASE_READ_PLL_DATA

***Function:*** Reads and returns the contents of the internal registers of the PLL.
***Input:*** None
***Output:*** LM9_BASE_PLL_DATA structure
***Notes:*** The PLL register data is returned in the LM9_BASE_PLL_DATA structure in an array of 40 bytes. See definition of LM9_BASE_PLL_DATA above.

### 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. See the Base Register Definitions section in the HW manual for exact bit definitions.

### 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. See the Base Register Definitions section in the HW manual for exact bit definitions.

### 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 the Base Register Definitions section in the HW manual for detailed description.

### 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. See the Base Register Definitions section in the HW manual for detailed description.

**IOCTL_LM9_BASE_GET_GPIO_TERM**

**Function:** Read from GPIO Termination Control Register
**Input:** none
**Output:** ULONG
**Notes**: See the Base Register Definitions section in the HW manual for detailed description.

**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 See the Base Register Definitions section in the HW manual for detailed description.

**IOCTL_LM9_BASE_GET_GPIO_DIR**

**Function:** Read from GPIO Direction Control Register
**Input:** none
**Output:** ULONG
**Notes:** See the Base Register Definitions section in the HW manual for detailed description.

**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. See the Base Register Definitions section in the HW manual for detailed description.

**IOCTL_LM9_BASE_GET_GPIO_DATA**

**Function:** Read from GPIO Data Control Register
**Input:** none
**Output:** ULONG
**Notes:** Read back of control register. See the Base Register Definitions section in the HW manual for detailed description.

### 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. See the Chan Bit Maps section in the HW manual for detailed description.

**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.

### 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 below.

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

### 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 the Chan Bit Maps section in the HW manual for exact bit definitions.

### 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. See the Chan Bit Maps section in the HW manual for exact bit definitions.

### 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. See the definition of the LM9_CHAN_FIFO_LEVELS structure below. See the Chan Bit Maps section in the HW manual for detailed description.

```
typedef struct _LM9_CHAN_FIFO_LEVELS{
       USHORT   AlmostFull; // Set to control RX Almost full interrupt definition
       USHORT   AlmostEmpty;// set to control TX Almost empty Interrupt defintion
} LM9_CHAN_FIFO_LEVELS, *PLM9_CHAN_FIFO_LEVELS;
```

### 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:** See the definition of the LM9_CHAN_FIFO_LEVELS structure above. See the Chan Bit Maps section in the HW manual for detailed description.

### 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. See the definition of the LM9_CHAN_FIFO_COUNTS structure below. See the Chan Bit Maps section in the HW manual for detailed description.

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

### IOCTL_LM9_CHAN_RESET_FIFOS

**Function:** Resets one or both internal FIFOs for the referenced channel.
**Input:** LM9_CHAN_FIFO_SEL enumeration type
**Output:** None
**Notes:** Resets Transmit, Receive, Both (Transmit and Receive) . See the definition of the LM9_CHAN_FIFO_SEL enumeration type below.

```
typedef enum _LM9_CHAN_FIFO_SEL {
       LM9_RX,
       LM9_TX,
       LM9_BOTH
} LM9_CHAN_FIFO_SEL, *PLM9_CHAN_FIFO_SEL;
```

### 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. See the Chan Bit Maps section in the HW manual for detailed description.

### 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. See the Chan Bit Maps section in the HW manual for detailed description.

### IOCTL_LM9_CHAN_SET_CONT

**Function:** write to Channel Control register using structure
**Input:** LM9_CHAN_CONT
**Output**: None
**Notes:** See below for the definition of the LM9_CHAN_CONT structure. See the Chan Bit Maps section in the HW manual for exact bit definitions.

```
typedef struct _LM9_CHAN_CONT{
       BOOLEAN       FifoTestEn;// BiPass Mode Control
       BOOLEAN       MIntEn;    // RX 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;
```

### IOCTL_LM9_CHAN_GET_CONT

**Function:** Read from Channel Control register using structure
**Input:** None
**Output**: LM9_CHAN_CONT
**Notes:** See above for the definition of the LM9_CHAN_CONT structure. See the Chan Bit Maps section in the HW manual for exact bit definitions.

### IOCTL_LM9_CHAN_SET_TX

**Function:** write to Channel Tx Control register using structure
**Input:** LM9_CHAN_TX_CONTROL
**Output**: None
**Notes:** See below for the definition of the LM9_CHAN_TX_CONTROL structure. See the Chan Bit Maps section in the HW manual for exact bit definitions.

```
typedef struct _LM9_CHAN_TX_CONTROL{
        BOOLEAN         TxStart;
        BOOLEAN         TxIntEn;
        BOOLEAN         TxAEIntEn;
        BOOLEAN         TxUnFlEn;
        BOOLEAN         TxByteOrder;
        BOOLEAN         TxBitOrder;
        BOOLEAN         TxClkPol;
        BOOLEAN         TxRegPacket;
        BOOLEAN         TxParity;
        BOOLEAN         TxClockDir;
        BOOLEAN         TxClockSrc;
        BOOLEAN         TxStartBit;
        BOOLEAN         TxMarkBit;
} LM9_CHAN_TX_CONTROL, *PLM9_CHAN_TX_CONTROL;
```

### IOCTL_LM9_CHAN_GET_TX

**Function:** Read from Channel Master Control register using structure
**Input:** None
**Output**: LM9_CHAN_TX_CONTROL
**Notes:** See above for the definition of the LM9_CHAN_CONT structure. See the Chan Bit Maps section in the HW manual for exact bit definitions.

### 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. See the Chan Bit Maps section in the HW manual for detailed description.

**IOCTL_LM9_CHAN_GET_TX_COUNT**

**Function:** Read from Channel TX Count Register
**Input:** None
**Output**: ULONG
**Notes:** See the Chan Bit Maps section in the HW manual for detailed description.

**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. See the Chan Bit Maps section in the HW manual for detailed description.

**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. See the Chan Bit Maps section in the HW manual for detailed description.

### IOCTL_LM9_CHAN_SET_RX

**Function:** write to Channel Receiver Control register using structure
**Input:** LM9_CHAN_RX_CONTROL
**Output**: None
**Notes:** See below for the definition of the LM9_CHAN_RX_CONTROL structure. See the Chan Bit Maps section in the HW manual for exact bit definitions.

```
typedef struct _LM9_CHAN_RX_CONTROL{
        BOOLEAN                                 RxStart;
        BOOLEAN                                 RxParityErrEn;
        BOOLEAN                                 RxIntEn;
        BOOLEAN                                 RxAFIntEn;
        BOOLEAN                                 RxOvFlEn;
        BOOLEAN                                 RxByteOrder;
        BOOLEAN                                 RxBitOrder;
        BOOLEAN                                 RxClkPol;
        BOOLEAN                                 RxParity;
        BOOLEAN                                 RxTimeOutEn;
        BOOLEAN                                 RxStartBit;
        BOOLEAN                                 RxMarkBit;
} LM9_CHAN_RX_CONTROL, *PLM9_CHAN_RX_CONTROL;
```

### IOCTL_LM9_CHAN_GET_RX

**Function:** Read from Channel Receiver Control register using structure
**Input:** None
**Output**: LM9_CHAN_RX_CONTROL
**Notes:** See above for the definition of the LM9_CHAN_RX_CONTROL structure. See the Chan Bit Maps section in the HW manual for exact bit definitions.

### 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. See the Chan Bit Maps section in the HW manual for detailed description.

### IOCTL_LM9_CHAN_GET_RX_COUNT

**Function:** Read from Channel Receiver Count register
**Input:** None
**Output**: ULONG
**Notes:** See the Chan Bit Maps section in the HW manual for detailed description.

### 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. See the Chan Bit Maps section in the HW manual for detailed description.

### 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. See the Chan Bit Maps section in the HW manual for detailed description.

### IOCTL_LM9_CHAN_RX_GET_TIMEOUT

**Function:** Read from Channel Receiver TimeOut Register
**Input:** None
**Output**: ULONG
**Notes:** See the Chan Bit Maps section in the HW manual for detailed description.

## 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

Please refer to the warranty page on our website for the current warranty offered and options.
http://www.dyneng.com/warranty.html

### 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 or e-mail and arrange to work with an engineer. We will work with you to determine the cause of the issue.

#### Support

The software described in this manual is provided at no cost to clients who have purchased the corresponding hardware. Minimal support is included along with the documentation. For help with integration into your project please contact sales@dyneng.com for a support contract. Several options are available. With a contract in place Dynamic Engineers can help with system debugging, special software development, or whatever you need to get going.

### 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