DYNAMIC ENGINEERING
150 DuBois, Suite 3
Santa Cruz, CA 95060
(831) 457-8891 **Fax** (831) 457-4793
http://www.dyneng.com
sales@dyneng.com
Est. 1988

PB3Lm6
&
Lm6Chan

Driver Documentation

Win32 Driver Model

Revision A
Corresponding Hardware: Revision C
10-2005-0203
Corresponding Firmware: Revision B/C

PB3Lm6, Lm6Chan
WDM Device Drivers for the
PMC-BiSerial-III-LM6
4-Channel PMC-Based Serial Interface

Dynamic Engineering
150 DuBois, Suite 3
Santa Cruz, CA 95060
(831) 457-8891
FAX: (831) 457-4793

Manual Revision A. Revised May 21, 2007

Table of Contents

Introduction

The PB3Lm6 and Lm6Chan drivers are Win32 driver model (WDM) device drivers for the PMC-BiSerial-III LM6 from Dynamic Engineering. The PMC-BiSerial-III board has a Spartan3-1500 Xilinx FPGA to implement the PCI interface, FIFOs and protocol control and status for four serial channels. Each channel has two 2k x 32-bit data FIFOs for data transmission and reception.

When the PMC-BiSerial-III LM6 is recognized by the PCI bus configuration utility it will start the PB3Lm6 driver. The PB3Lm6 driver enumerates the channels and creates four separate Lm6Chan device objects. This allows the I/O channels to be totally independent while the base driver controls the device items that are common. 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 I/O channel devices.

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 PMC-BiSerial-III LM6 user manual (also referred to as the hardware manual).

Driver Installation

There are several files provided in each driver package. These files include PB3Lm6.sys, PB3Lm6.inf, DDPB3Lm6.h, PB3Lm6GUID.h, Lm6Chan.sys, Lm6Chan.inf, DDLm6Chan.h, Lm6ChanGUID.h, PB3Lm6Test.exe, and PB3Lm6Test source files. The DDPB3Lm6.h and DDLm6Chan.h files are C header files that define the Application Program Interface (API) to the drivers. The PB3Lm6GUID.h and Lm6ChanGUID.h files are C header files that define the device interface identifiers for the PB3Lm6 and Lm6Chan 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 PB3Lm6Test.exe file is a sample Win32 console application that makes calls into the PB3Lm6/Lm6Chan drivers to test each driver call without actually writing any application code. It is not required during the driver installation.

To run PB3Lm6Test.exe, open a command prompt console window and type a command. Type ***PB3Lm6Test -d0 -?*** to display a list of commands (the PB3Lm6Test.exe file must be in the directory that the window is referencing). The commands are all of the form ***PB3Lm6Test -dn -im*** where ***n*** and ***m*** are the device number and driver PB3Lm6 ioctl number respectively or ***PB3Lm6Test -cn -im*** where ***n*** and ***m*** are the channel number and Lm6Chan driver ioctl number respectively. This application is intended to test the proper functioning of the driver calls, not for normal operation.

Windows 2000 Installation

Copy PB3Lm6.inf, Lm6Chan.inf, PB3Lm6.sys and Lm6Chan.sys to a floppy disk, or CD if preferred.

With the PMC-BiSerial-III LM6 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 PB3Lm6.inf file.
- Select ***Next***.
- Select ***Finish*** to close the ***Found New Hardware Wizard***.

The system should now see the PB3Lm6 channels and reopen the ***New Hardware Wizard.*** Proceed as above substituting Lm6Chan.inf for PB3Lm6.inf.

Windows XP Installation

Copy PB3Lm6.inf, Lm6Chan.inf, PB3Lm6.sys and Lm6Chan.sys to a floppy disk, or CD if preferred.

With the PMC-BiSerial-III LM6 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 Updat***e***.
- Select ***Next***.
- Select ***Install the software automatically***.
- Select ***Next***.
- Select ***Finish*** to close the ***Found New Hardware Wizard***.

The system should now see the PB3Lm6 channel and reopen the ***New Hardware Wizard***. Proceed as above for each channel driver as necessary.

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 a globally unique identifier (GUID), which is defined in PB3Lm6GUID.h and Lm6ChanGUID.h.

Below is example code for opening handles for device *devNum*.

```
// Maximum length of the device name for a given interface
#define MAX_DEVICE_NAME 256

// Handles to device objects
HANDLE hPB3Lm6                          =  INVALID_HANDLE_VALUE;
HANDLE hLm6Chan[PB3_LM6_NUM_CHANNELS] = {INVALID_HANDLE_VALUE,
                                         INVALID_HANDLE_VALUE,
                                         INVALID_HANDLE_VALUE,
                                         INVALID_HANDLE_VALUE};

// PB3Lm6 device number
ULONG                           devNum;

// Lm6 channel handle array index and instance number
ULONG                           chan, chanDev;

// Return status from command
LONG                            status;

// Handle to device interface information structure
HDEVINFO                        hDeviceInfo;

// The actual symbolic link name to use in the CreateFile() call
CHAR                            deviceName[MAX_DEVICE_NAME];

// Size of buffer reguired 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_PB3_LM6,
                        NULL,
                        NULL,
                        DIGCF_PRESENT | DIGCF_DEVICEINTERFACE);
```

```
if(hDeviceInfo == INVALID_HANDLE_VALUE)
{
   status = GetLastError();
   printf("**Error: couldn't get class info, (%d)\n", status);
   exit(-1);
}

interfaceData.cbSize = sizeof(interfaceData);

// Find the interface for device devNum
if(!SetupDiEnumDeviceInterfaces(hDeviceInfo,
                                NULL,
                        (LPGUID)&GUID_DEVINTERFACE_PB3_LM6,
                                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);
   }
}

// Found our device-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
hPB3Lm6 = CreateFile(deviceName,
                     GENERIC_READ    | GENERIC_WRITE,
                     FILE_SHARE_READ | FILE_SHARE_WRITE,
                     NULL,
                     OPEN_EXISTING,
                     NULL,
                     NULL);

if(hPB3Lm6 == INVALID_HANDLE_VALUE)
{
   printf("**Error: couldn't open %s, (%d)\n", deviceName, GetLastError());
   exit(-1);
}

hDeviceInfo = SetupDiGetClassDevs(
                (LPGUID)&GUID_DEVINTERFACE_LM6_CHAN,
                        NULL,
                        NULL,
                        DIGCF_PRESENT | DIGCF_DEVICEINTERFACE);
```

```
if(hDeviceInfo == INVALID_HANDLE_VALUE)
{
   status = GetLastError();
   printf("**Error: couldn't get class info, (%d)\n", status);
   exit(-1);
}

interfaceData.cbSize = sizeof(interfaceData);

for(chan = 0, chanDev = devNum * PB3_LM6_NUM_CHANNELS;
    chan < PB3_LM6_NUM_CHANNELS;
    chan++, chanDev++)
{// Find the interface for chanDev device
   if(!SetupDiEnumDeviceInterfaces(hDeviceInfo,
                                   NULL,
                           (LPGUID)&GUID_DEVINTERFACE_LM6_CHAN,
                                    chanDev,
                                   &interfaceData))
   {
      status = GetLastError();
      if(status == ERROR_NO_MORE_ITEMS)
      {
         printf("**Error: couldn't find device(no more items), (%d)\n",
                 chanDev);

         SetupDiDestroyDeviceInfoList(hDeviceInfo);
         exit(-1);
      }
      else
      {
         printf("**Error: couldn't enum device, (%d)\n", status);
         SetupDiDestroyDeviceInfoList(hDeviceInfo);
         exit(-1);
      }
   }

// Found our device-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);

// Open driver - Create the handle to the device
   hLm6Chan[chan] = CreateFile(deviceName,
                               GENERIC_READ    | GENERIC_WRITE,
                               FILE_SHARE_READ | FILE_SHARE_WRITE,
                               NULL,
                               OPEN_EXISTING,
                               NULL, (or FILE_FLAG_OVERLAPPED for async I/O)
                               NULL);

   if(hLm6Chan[chan] == INVALID_HANDLE_VALUE)
   {
      printf("**Error: couldn't open %s, (%d)\n",
              deviceName,
              GetLastError());

      SetupDiDestroyDeviceInfoList(hDeviceInfo);
      exit(-1);
   }
}
// Cleanup search
SetupDiDestroyDeviceInfoList(hDeviceInfo);
```

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 or channel. 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 the lm6 drivers are described below:

IOCTL_PB3_LM6_GET_INFO

Function: Returns the device driver version, Xilinx flash revision, PLL device ID, user switch value, and instance number.
Input: None
Output: PB3_LM6_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 DDPB3Lm6.h for the definition of PB3_LM6_DRIVER_DEVICE_INFO.

IOCTL_PB3_LM6_LOAD_PLL_DATA

Function: Writes to the internal registers of the PLL.
Input: PB3_LM6_PLL_DATA structure
Output: None
Notes: The PB3_LM6_PLL_DATA structure has only one field: Data – an array of 40 bytes containing the PLL register data to write.

IOCTL_PB3_LM6_READ_PLL_DATA

Function: Returns the contents of the internal registers of the PLL.
Input: None
Output: PB3_LM6_PLL_DATA structure
Notes: The register data is written to the PB3_LM6_PLL_DATA structure in an array of 40 bytes.

IOCTL_LM6_CHAN_GET_INFO

Function: Returns the device driver version and the channel instance number.
Input: None
Output: LM6_CHAN_DRIVER_DEVICE_INFO structure
Notes: See DDPB3Lm6.h for the definition of LM6_CHAN_DRIVER_DEVICE_INFO.

IOCTL_LM6_CHAN_RESET_FIFOS

Function: Resets the Tx and/or Rx FIFOs for the referenced channel.
Input: LM6_FIFO_SEL enumerated type
Output: None
Notes: Resets the Transmit and/or Receive FIFOs for the referenced channel.

IOCTL_LM6_CHAN_SEND_FRAME

Function: Send a data-frame provided sufficient data is in the FIFO.
Input: Number of packets to send (unsigned short integer)
Output: None
Notes: The test transmitter on channels two and three ignores the packet count and sends a single stream of packets until the FIFO contains insufficient data to continue.

IOCTL_LM6_CHAN_STOP_FRAME

Function: Abort or cancel a data-frame.
Input: None
Output: None
Notes: This call will cancel a send request that has not started or stop a transmission in progress after the current packet completes.

IOCTL_LM6_CHAN_SET_CONFIG

Function: Writes to the channel's control register.
Input: Value of the control register (unsigned long integer)
Output: None
Notes: Only the bits in the CNTRL_MASK are controlled by this command. See the bit definitions in DDLm6Chan.h for information on determining this value.

IOCTL_LM6_CHAN_GET_CONFIG

Function: Returns the configuration of the control register.
Input: None
Output: Value of control register (unsigned long integer)
Notes: The return value includes the bits in CNTRL_MASK plus CNTRL_DMA_WREN, CNTRL_DMA_RDEN, CNTRL_MINTEN and the current packet count value. This command is used mainly for testing.

IOCTL_LM6_CHAN_GET_STATUS

Function: Returns the channel's status value and clears the latched status bits.
Input: None
Output: Value of the channel's status register (unsigned long integer)
Notes: See DDLm6Chan.h for the status bit definitions. Only the bits in STATUS_MASK will be returned. The bits in STATUS_LATCH_MASK will be cleared by this call only if they are set when the register was read. This prevents the possibility of missing an interrupt condition that occurs after the register has been read but before the register write that clears the bits.

IOCTL_LM6_CHAN_GET_ERROR_COUNT

Function: Returns the value of the parity error counter and clears the count.
Input: None
Output: Parity error count (unsigned short integer)
Notes: Each channel contains a 16-bit counter that increments each time a parity error is detected. If the counter reaches its maximum count (65,535), it will hold that value until it has been read regardless of how many errors are subsequently detected. On channels two and three only the test receiver packets include parity, so this count is not relevant for the operational receiver on those channels.

IOCTL_LM6_CHAN_SET_FIFO_LEVELS

Function: Sets the channel's receiver almost full and transmitter almost empty levels.
Input: LM6_CHAN_FIFO_LEVELS structure
Output: None
Notes: These FIFO levels are used to determine status when the FIFO data counts reach the specified levels. They are also used to signal priority for the DMA request/grant arbiter, if this has been enabled for the referenced channel. See DDLm6Chan.h for the definition of LM6_CHAN_FIFO_LEVELS.

IOCTL_LM6_CHAN_GET_FIFO_LEVELS

Function: Returns the channel's receiver almost full and transmitter almost empty levels.
Input: None
Output: LM6_CHAN_FIFO_LEVELS structure
Notes: See DDLm6Chan.h for the definition of LM6_CHAN_FIFO_LEVELS.

IOCTL_LM6_CHAN_WRITE_FIFO

Function: Writes a single data word to the channel's transmit FIFO.
Input: FIFO data word (unsigned long integer)
Output: None
Notes: Normally the write command is used to load data into the device. This call can be used for small amounts of data, but is inefficient for larger sized transfers.

IOCTL_LM6_CHAN_READ_FIFO

Function: Reads a single data word from the channel's receive FIFO.
Input: None
Output: FIFO data word (unsigned long integer)
Notes: Normally the read command is used to retrieve data from the device. This call can be used for small amounts of data, but is inefficient for larger sized transfers.

IOCTL_LM6_CHAN_GET_FIFO_COUNTS

Function: Returns the number of data words in the transmit and receive FIFOs.
Input: None
Output: LM6_CHAN_FIFO_COUNTS structure
Notes: Returns the number of words in the referenced channels I/O data circuitry. For the transmitter this is a maximum of one more than the FIFO size and for the receiver the data-count can be as much as four words more than the FIFO size. The excess is due to data pipe-line latches in the I/O data-path. See DDLm6Chan.h for the definition of LM6_CHAN_FIFO_COUNTS.

IOCTL_LM6_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_LM6_CHAN_ENABLE_INTERRUPT

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

IOCTL_LM6_CHAN_DISABLE_INTERRUPT

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

IOCTL_LM6_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 if the master interrupt is enabled. This IOCTL is used for test and development, to test interrupt processing.

IOCTL_LM6_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 for the last user interrupt serviced.

Write

PMC-BiSerial-III LM6 DMA data is written to the device using the write command. Writes are executed using the Win32 function WriteFile() and passing in the handle to the device opened with CreateFile(), a pointer to a pre-allocated buffer containing the data to be written, an unsigned long integer that represents the size of that buffer in bytes, a pointer to an unsigned long integer to contain the number of bytes actually written, and a pointer to an optional Overlapped structure for performing asynchronous I/O.

Read

PMC-BiSerial-III LM6 DMA data is read from the device using the read command. Reads are executed using the Win32 function ReadFile() and passing in the handle to the device opened with CreateFile(), a pointer to a pre-allocated buffer that will contain the data read, an unsigned long integer that represents the size of that buffer in bytes, a pointer to an unsigned long integer to contain the number of bytes actually read, and a pointer to an optional Overlapped structure for performing asynchronous I/O.

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, Suite 3
Santa Cruz, CA 95060
(831) 457-8891 Fax (831) 457-4793
support@dyneng.com