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

User Manual

PMC BiSerial-II NG1
Driver Documentation

Revision B
Corresponding Hardware: Revision A
10-2002-1201

PMC BiSerial-II NG1
Bi-directional Serial Data Interface PMC Module

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

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.

This product has been designed to operate with PMC Module carriers and compatible user-provided equipment. Connection of incompatible hardware is likely to cause serious damage.

Manual Revision B. Revised May 20, 2003.

Table of Contents

Introduction

The PB2_NG1 driver is a Windows 2000 driver for the PMC BiSerial-II NG1 board from Dynamic Engineering. Each PMC BiSerial-II NG1 board transmits and receives two channels of serial data over full-duplex UART interfaces, as well as two half-duplex Index channels with EIA-RS-485 differential drivers and receivers. A separate "Device Object" controls each PMC BiSerial-II NG1 board, and a separate handle references each Device Object. IO Control calls (IOCTLs) are used to configure the hardware and transfer data to and from the device over the PCI bus.

Note

This documentation will provide information about all calls made to the driver, and how the driver interacts with the device for each of these calls. For more detailed information on the hardware implementation, refer to the PMC BiSerial-II NG1 device user manual.

Driver Installation

There are several files provided in each driver drop. These files include PB2_NG1.sys, PB2_NG1.inf, DDPB2_NG1.h, PB2_NG1Test.exe, and PB2_NG1Test source files.

When the Windows2000 system sees the hardware for the first time it will start the *Found New Hardware Wizard*. Click on next and select *Search for a suitable driver for my device*. Click on next and specify a location that contains the *PB2_NG1.sys* and *PB2_NG1.inf* files. Follow the prompts until the Wizard finishes and the driver is installed.

The DDPB2_NG1.h file is the C header file that defines the Application Program Interface (API) to the driver. This file is required at compile time by any application that wishes to interface with the PMC BiSerial-II NG1 device. It is not needed by the driver installation.

The PB2_NG1test.exe file is a sample Windows 2000 console application that makes calls into the PB2_NG1 driver to test the driver calls without actually writing an application. It is not required during the driver installation.

Driver Startup

In Windows2000 once the driver has been installed it will start automatically when it sees 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 DDPB2_NG1.h.

Below is example code for opening a handle for device 0. The device number is underlined and italicized in the `SetupDiEnumDeviceInterfaces` call.

```
#define MAX_DEVICE_NAME 256 // the maximum length of the device name for
                            //  a given instance of an interface

HANDLE   hPb2_ng1 // Handle to the device object
// 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_PB2_NG1,
                                  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 0
if(!SetupDiEnumDeviceInterfaces(hDeviceInfo,
                                NULL,
                                (LPGUID)&GUID_DEVINTERFACE_PB2_NG1,
                                0,
                                &interfaceData))
{
   status = GetLastError();
```

```
   if(status == ERROR_NO_MORE_ITEMS)
   {
      printf("**Error: couldn't find device(no more items), (%d)\n", 0);
      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
hPb2_ng1 = CreateFile(deviceName,
                      GENERIC_READ    | GENERIC_WRITE,
                      FILE_SHARE_READ | FILE_SHARE_WRITE,
                      NULL,
                      OPEN_EXISTING,
                      NULL,
                      NULL);
```

IO Controls

The driver uses IO Control calls (IOCTLs) to configure the device. IOCTLs refer to a single Device Object in the driver, 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.

IOCTL_PB2_NG1_GET_VERSION

Function: Returns the driver and Xilinx version information.
Input: none
Output: ULONG
Notes: The Xilinx version is encoded in bits 8 to 15 and the driver version is encoded in bits 0 to 7.

IOCTL_PB2_NG1_GET_SW_ID

Function: Returns the user switch value.
Input: none
Output: ULONG
Notes: Returns the value set in the eight-position DIP switch on the PMC BiSerial-II NG1.

IOCTL_PB2_NG1_GET_STATUS0

Function: Returns the board status.
Input: none
Output: ULONG
Notes: Returns Status information for a given board obtained from the PB2_NG1_STAT0 register. This includes FIFO flags indicating the amount of data in FIFOs A and B and response FIFOs 1 and 2, the state of the static RS-485 lines when these are configured as inputs, and the interrupt status bit, which indicates that an enabled interrupt condition is active. See the bit definitions in the DDPB2_NG1.h header file for more information.

IOCTL_PB2_NG1_GET_STATUS1

Function: Returns the board status.
Input: none
Output: ULONG
Notes: Returns Status information for a given board obtained from the PB2_NG1_STAT1 register. This consists of latched interrupt status bits indicating the cause of an interrupt. After the status is read, a value is written back to this register to clear only the specific interrupt conditions that were read. This will insure that no interrupt cause is missed due to being asserted between the read and write cycles. See the bit definitions in the DDPB2_NG1.h header file for more information.

IOCTL_PB2_NG1_SET_BASE_CONFIG

Function: Sets the base configuration of the board.
Input: ULONG
Output: none
Notes: Controls the clock source and divisor for determining the reference clock frequency and enables the internal clock as well as the two external clock outputs. Also sets the direction of the static RS-485 signals, the output values of these when they are configured as outputs, and the FIFO A and B loop-back path enables. See the bit definitions in the DDPB2_NG1.h header file for more information.

IOCTL_PB2_NG1_GET_BASE_CONFIG

Function: Returns the base configuration of the board.
Input: none
Output: ULONG
Notes: Returns the base configuration register value, excluding the FIFO Id and enable bits. See the bit definitions in the DDPB2_NG1.h header file for more information.

IOCTL_PB2_NG1_SET_INTEN_CONFIG

Function: Sets the interrupt enable configuration.
Input: ULONG
Output: none
Notes: Controls the individual interrupt enables for all sixteen interrupt conditions.

IOCTL_PB2_NG1_GET_INTEN_CONFIG

Function: Returns the interrupt enable configuration.
Input: none
Output: ULONG
Notes: Returns the individual enable states of the sixteen possible interrupt conditions.

IOCTL_PB2_NG1_SET_UART1_CONFIG

Function: Sets the configuration of the UART1 interface.
Input: ULONG
Output: none
Notes: Controls all the functions of the UART1 interface except the LRU identification fields. See the bit definitions in the DDPB2_NG1.h header file for more information.

IOCTL_PB2_NG1_GET_UART1_CONFIG

Function: Returns the UART1 configuration.
Input: none
Output: ULONG
Notes: Returns the state of the UART1 configuration. See the bit definitions in the DDPB2_NG1.h header file for more information.

IOCTL_PB2_NG1_SET_UART2_CONFIG

Function: Sets the configuration of the UART2 interface.
Input: ULONG
Output: none
Notes: Controls all the functions of the UART2 except the LRU identification fields. See the bit definitions in the DDPB2_NG1.h header file for more information.

IOCTL_PB2_NG1_GET_UART2_CONFIG

Function: Returns the UART2 configuration.
Input: none
Output: ULONG
Notes: Returns the state of the UART2 configuration. See the bit definitions in the DDPB2_NG1.h header file for more information.

IOCTL_PB2_NG1_SET_INDEX1_CONFIG

Function: Sets the configuration of the Index1 interface.
Input: ULONG
Output: none
Notes: Controls all the functions of the Index1 interface. See the bit definitions in the DDPB2_NG1.h header file for more information.

IOCTL_PB2_NG1_GET_INDEX1_CONFIG

Function: Returns the Index1 configuration.
Input: none
Output: ULONG
Notes: Returns the state of the Index1 configuration. See the bit definitions in the DDPB2_NG1.h header file for more information.

IOCTL_PB2_NG1_SET_INDEX2_CONFIG

Function: Sets the configuration of the Index2 interface.
Input: ULONG
Output: none
Notes: Controls all the functions of the Index2 interface. See the bit definitions in the DDPB2_NG1.h header file for more information.

IOCTL_PB2_NG1_GET_INDEX2_CONFIG

Function: Returns the Index2 configuration.
Input: none
Output: ULONG
Notes: Returns the state of the Index2 configuration. See the bit definitions in the DDPB2_NG1.h header file for more information.

IOCTL_PB2_NG1_SET_TERMINATIONS

Function: Sets the configuration of the driver terminations.
Input: ULONG
Output: none
Notes: Sets the configuration of the terminations for the IO lines. See the bit definitions in the DDPB2_NG1.h header file for more information.

IOCTL_PB2_NG1_GET_TERMINATIONS

Function: Returns the configuration of the driver terminations.
Input: none
Output: ULONG
Notes: Returns the configuration of the terminations for the IO lines. See the bit definitions in the DDPB2_NG1.h header file for more information.

IOCTL_PB2_NG1_SET_FIFO_LEVELS

Function: Sets FIFO A and B almost full and almost empty levels.
Input: FIFO_LEVELS
Output: none
Notes: Sets the almost full level for both FIFOs; the value is the number of words below full that the PAF flag becomes asserted. Sets the almost empty level for the FIFOs; the value is the number of words above empty for which the PAE flag is asserted. The UART state machines are stopped by this command, since normal FIFO data accesses are disabled when these level registers are accessed. Values are checked to not exceed the FIFO sizes.

IOCTL_PB2_NG1_GET_FIFO_LEVELS

Function: Returns almost full and almost empty FIFO levels.
Input: none
Output: FIFO_LEVELS
Notes: Returns the almost full and the almost empty level for FIFO A and B. The UART state machines are stopped by this command, since normal FIFO data accesses are disabled when these level registers are accessed.

IOCTL_PB2_NG1_RESET_FIFOS

Function: Resets the transmit and receive FIFOs.
Input: FIFO_RESETS
Output: none
Notes: Resets either or both FIFOs, as indicated in the FIFO_RESETS structure. This will clear all data and reset the almost full and empty values to their default values.

IOCTL_PB2_NG1_LOAD_FIFO_A

Function: Loads one 32-bit word into FIFO A, if it is configured to accept data.
Input: ULONG
Output: none
Notes: The FIFO must be enabled, and UART channel 1 must be configured as a controller, or the FIFO loop test enabled.

IOCTL_PB2_NG1_READ_FIFO_A

Function: Reads one 32-bit word from FIFO A, if it is configured to retrieve data.
Input: none
Output: ULONG
Notes: The FIFO must be enabled, and UART channel 1 must be configured as a terminal, or the FIFO loop test enabled.

IOCTL_PB2_NG1_LOAD_UART1_RD_DATA

Function: Loads a 16-bit word into the UART 1 read response data register.
Input: ULONG
Output: none
Notes: The data overwrites any data that was previously written to this register. This data will be returned in the data field of the response to a read request packet.

IOCTL_PB2_NG1_READ_UART1_RD_DATA

Function: Returns the data currently loaded into the read response data register.
Input: none
Output: ULONG
Notes: This data is the last 16-bit data word written to the register. This call is mainly for test purposes.

IOCTL_PB2_NG1_READ_UART1_RSP_PACKET

Function: Returns an acknowledge packet from response FIFO 1.
Input: none
Output: ACK_PACKET
Notes: This call will return an ACK_PACKET structure containing three fields. The second field is the length of the packet in bytes, this should be two for an acknowledge1, or five for an acknowledge2. The third field is an array of bytes that contains the packet data. The first field is a BOOLEAN value, *valid*, that is true if sufficient bytes were available to satisfy the packet length encoded in the packet size field. At most five bytes will be read from the FIFO.

IOCTL_PB2_NG1_LOAD_FIFO_B

Function: Loads one 32-bit word into FIFO B, if it is configured to accept data.
Input: ULONG
Output: none
Notes: The FIFO must be enabled, and UART channel 2 must be configured as a controller, or the FIFO B loop test enabled.

IOCTL_PB2_NG1_READ_FIFO_B

Function: Reads one 32-bit word from FIFO B, if it is configured to retrieve data.
Input: none
Output: ULONG
Notes: The FIFO must be enabled, and UART channel 2 must be configured as a terminal, or the FIFO loop test enabled.

IOCTL_PB2_NG1_LOAD_UART2_RD_DATA

Function: Loads a 16-bit word into the UART 2 read response data register.
Input: ULONG
Output: none
Notes: The data overwrites any data that was previously written to this register. This data will be returned in the data field of the response to a read request packet.

IOCTL_PB2_NG1_READ_UART2_RD_DATA

Function: Returns the data currently loaded into the read response data register.
Input: none
Output: ULONG
Notes: This data is the last 16-bit data word written to the register. This call is mainly for test purposes.

IOCTL_PB2_NG1_READ_UART2_RSP_PACKET

Function: Returns an acknowledge packet from response FIFO 2.
Input: none
Output: ACK_PACKET
Notes: This call will return an ACK_PACKET structure containing three fields. The second field is the length of the packet in bytes, this should be two for an acknowledge1, or five for an acknowledge2. The third field is an array of bytes that contains the packet data. The first field is a BOOLEAN value, *valid*, that is true if sufficient bytes were available to satisfy the packet length encoded in the packet size field. At most five bytes will be read from the FIFO.

IOCTL_PB2_NG1_LOAD_INDEX1

Function: Loads a nine-bit word into the Index 1 data register.
Input: ULONG
Output: none
Notes: If Index channel 1 is enabled and auto-transmit is not enabled, the data written to this register is shifted out immediately with start and stop bits added.

IOCTL_PB2_NG1_READ_INDEX1

Function: Reads a nine or twelve-bit response word from Index channel 1.
Input: none
Output: ULONG
Notes: If the Index 1 data available bit is set, valid data will be read from this data register.

IOCTL_PB2_NG1_LOAD_INDEX1_TX1

Function: Loads a nine-bit data word into the first auto-data register for Index 1.
Input: ULONG
Output: none
Notes: This data will be sent by Index channel 1 when the auto-transmit mode is enabled and the Tx1 register is enabled.

IOCTL_PB2_NG1_READ_INDEX1_TX1

Function: Reads the nine-bit data word last written to the Index 1 Tx1 register.
Input: none
Output: ULONG
Notes: This call is mainly used for test purposes.

IOCTL_PB2_NG1_LOAD_INDEX1_TX2

Function: Loads a nine-bit data word into the second auto-transmit register for Index 1.
Input: ULONG
Output: none
Notes: This data will be sent by Index channel 1 when the auto-transmit mode is enabled and the Tx2 register is enabled.

IOCTL_PB2_NG1_READ_INDEX1_TX2

Function: Reads the nine-bit data word last written to the Index 1 Tx2 register.
Input: none
Output: ULONG
Notes: This call is mainly used for test purposes.

IOCTL_PB2_NG1_LOAD_INDEX2

Function: Loads a nine-bit word into the Index 2 data register.
Input: ULONG
Output: none
Notes: If Index channel 2 is enabled and auto-transmit is not enabled, the data written to this register is shifted out immediately with start and stop bits added.

IOCTL_PB2_NG1_READ_INDEX2

Function: Reads a nine or twelve-bit response word from Index channel 2.
Input: none
Output: ULONG
Notes: If the Index 2 data available bit is set, valid data will be read from this data register.

IOCTL_PB2_NG1_LOAD_INDEX2_TX1

Function: Loads a nine-bit data word into the first auto-data register for Index 2.
Input: ULONG
Output: none
Notes: This data will be sent by Index channel 2 when the auto-transmit mode is enabled and the Tx1 register is enabled.

IOCTL_PB2_NG1_READ_INDEX2_TX1

Function: Reads the nine-bit data word last written to the Index 2 Tx1 register.
Input: none
Output: ULONG
Notes: This call is mainly used for test purposes.

IOCTL_PB2_NG1_LOAD_INDEX2_TX2

Function: Loads a nine-bit data word into the second auto-transmit register for Index 2.
Input: ULONG
Output: none
Notes: This data will be sent by Index channel 2 when the auto-transmit mode is enabled and the Tx2 register is enabled.

IOCTL_PB2_NG1_READ_INDEX2_TX2

Function: Reads the nine-bit data word last written to the Index 2 Tx2 register.
Input: none
Output: ULONG
Notes: This call is mainly used for test purposes.

IOCTL_PB2_NG1_REGISTER_EVENT

Function: Registers an event to be signaled when an interrupt occurs.
Input: Event Handle
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 an interrupt is serviced. The user interrupt service routine waits on this event, allowing it to respond to the interrupt.

IOCTL_PB2_NG1_ENABLE_INTERRUPT

Function: Sets the master interrupt enable to true.
Input: none
Output: none
Notes: Sets the master interrupt enable, leaving all other bit values in the interrupt enable configuration register the same. This IOCTL is used in the user interrupt processing function to re-enable the interrupts after they were disabled in the driver interrupt service routine. This allows that function to enable the interrupts without knowing the particulars of the other configuration bits.

IOCTL_PB2_NG1_DISABLE_INTERRUPT

Function: Sets the master interrupt enable to true.
Input: none
Output: none
Notes: Clears the master interrupt enable, leaving all other bit values in the interrupt enable configuration register the same. This IOCTL is used when interrupt processing is no longer desired.

IOCTL_PB2_NG1_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 enable is set. This IOCTL is used for development, to test interrupt processing.

IOCTL_PB2_NG1_LOAD_UART1_LRU_ID

Function: Loads the five LRU ID fields in the UART1 configuration register.
Input: LRU_ID
Output: none
Notes: The LRU_ID structure contains a NumIDs field that specifies how many IDs are to be loaded and an array of five bytes that contains the LRU IDs. Only the number of IDs indicated need to be written into the LRU_ID struct e.g. if there is only one valid LRU ID, write a one in the NumIDs field and enter the LRU ID number in the first position of the IdData array. When the IOCTL is executed the driver will automatically fill all the ID fields with that ID number. This allows the UART to check from one to five ID numbers to determine the state of the ID_ERR bit in the acknowledge packet. See the DDPB2_NG1.h header file for more information and the LRU_ID definition.

IOCTL_PB2_NG1_LOAD_UART2_LRU_ID

Function: Loads the five LRU ID fields in the UART2 configuration register.
Input: LRU_ID
Output: none
Notes: The LRU_ID structure contains a NumIDs field that specifies how many IDs are to be loaded and an array of five bytes that contains the LRU IDs. Only the number of IDs indicated need to be written into the LRU_ID struct e.g. if there is only one valid LRU ID, write a one in the NumIDs field and enter the LRU ID number in the first position of the IdData array. When the IOCTL is executed the driver will automatically fill all the ID fields with that ID number. This allows the UART to check from one to five ID numbers to determine the state of the ID_ERR bit in the acknowledge packet. See the DDPB2_NG1.h header file for more information and the LRU_ID definition.

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