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

IP-BiSerial-Miller
Driver Documentation

Revision A
Corresponding Hardware: Revision A

IP-BiSerial-Miller
Bi-directional Serial Data Interface
IP Module

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

Manual Revision A. Revised October 31, 2003.

Table of Contents

Introduction

The IpBiS_Mlr driver is a WindowsXP driver for the IP-BiSerial-Miller [-Mlr] board from Dynamic Engineering. Each IP-BiSerial-Mlr board transmits and receives one channel of serial data with both TTL and EIA-RS-485 differential drivers and receivers. A separate “Device Object” controls each IP-BiSerial-Mlr board, and a separate handle references each Device Object. IO Control calls (IOCTLs) are used to configure the hardware and ReadFile() and WriteFile() calls are used to transfer data to and from the device over the IP 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 IP-BiSerial-Mlr device user manual.

Driver Installation

There are several files provided in each driver drop. These files include IpBiS_Mlr.sys, IpDev.inf, DDIpBiS_Mlr.h, IpBiS_MlrGUID.h, IpBiS_MlrDef.h, BSMTest.exe, and BSMTest source files.

Copy IpDev.inf to the WINDOWS\INF folder and copy IpBiS_Mlr.sys to a floppy disk, or CD if preferred. Right click on the IpDev.inf file icon in the WINDOWS\INF folder and select *Install* from the pop-up menu. This will create a precompiled information file (.pnf) in the same directory.

When the WindowsXP system sees the hardware for the first time it will start the *Found New Hardware Wizard* (if the IP carrier driver has not been installed, the system will be unable to see any IP devices). The IP-BiSerial-Mlr should be named in the dialogue box. Follow the steps below:

- Insert the disk prepared above in the appropriate drive.
- Select *Install from a list or specific location*
- Select *Next*
- Select *Don't search. I will choose the driver to install*
- Select *Next*
- Select *Show all devices* from the list
- Select *Next*
- Select *Dynamic Engineering* from the Manufacturer list
- Select *IP-BiSerial-Mlr Device* from the Model list
- Select *Next*
- Select *Yes* on the Update Driver Warning dialogue box.
- Enter the drive *e.g. A:* in the *Files Needed* dialogue box.
- Select *OK*.
- Select *Finish* to close the *Found New Hardware Wizard*.

This process must be completed for each new device that is installed.

The DDIpBiS_Mlr.h file is the C header file that defines the Application Program Interface (API) to the driver. The IpBiS_MlrGUID.h file is a C header file that defines the device interface identifier for the IpBiS_Mlr. These files are required at compile time by any application that wishes to interface with the IP-BiSerial-Mlr driver. The IpBiS_MlrDef.h file contains the relevant bit defines for the IP-BiSerial-Mlr registers. These files are not needed for driver installation.

The BSMTest.exe file is a sample WindowsXP console application that makes calls into the IpBiS_Mlr driver to test the driver calls without actually writing an application. It is not required during the driver installation. Open a command prompt console window and type *BSMTest –dO -?* to display a list of commands (the BSMTest.exe file must be in the directory that the window is referencing). The commands are all of the form *BSMTest –dn –im* where *n* and *m* are the device number and driver ioctl number respectively.

Driver Startup

In WindowsXP 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 IpBiS_MlrGUID.h.

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

```
// The maximum length of the device name for
//  a given instance of an interface
#define MAX_DEVICE_NAME 256
// Handle to the device object
HANDLE                           hIpBiSMlr = INVALID_HANDLE_VALUE;
// Return status from command
LONG                             status;
// Handle to device interface information structure
HDEVINFO                         hDeviceInfo;
// The actual symbolic link name to use in the createfile
CHAR                             deviceName[MAX_DEVICE_NAME];
// Size of buffer required to get the symbolic link name
DWORD                            requiredSize;
// Interface data structures for this device
SP_DEVICE_INTERFACE_DATA         interfaceData;
PSP_DEVICE_INTERFACE_DETAIL_DATA pDeviceDetail;

hDeviceInfo = SetupDiGetClassDevs((LPGUID)&GUID_DEVINTERFACE_IPBIS_MLR,
                                  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_IPBIS_MLR,
                                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
hIpBiSMlr = CreateFile(deviceName,
                       GENERIC_READ    | GENERIC_WRITE,
                       FILE_SHARE_READ | FILE_SHARE_WRITE,
                       NULL,
                       OPEN_EXISTING,
                       NULL,
                       NULL);

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

IO Controls

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

Function: Returns the current driver version and Tx, Rx FIFO sizes.
Input: none
Output: DRIVER_IP_DEVICE_INFO struct
Notes: This call does not access the hardware, only driver parameters. See DDIpBiS_Mlr.h for the definition of DEVICE_IP_DEVICE_INFO.

IOCTL_IPBIS_MLR_GET_TX_DATA

Function: Reads one word from the Tx FIFO.
Input: none
Output: USHORT
Notes: This call is used for transmit FIFO write/read-back testing and is not needed for normal operation.

IOCTL_IPBIS_MLR_PUT_RX_DATA

Function: Writes one word to the Rx FIFO.
Input: USHORT
Output: none
Notes: This call is used for receive FIFO write/read-back testing and is not needed for normal operation.

IOCTL_IPBIS_MLR_GET_STATUS0

Function: Returns the FIFO status.
Input: none
Output: USHORT
Notes: Returns Status information for a given board obtained from the Status 0 register. This consists of the FIFO flags indicating the amount of data in the Tx and Rx FIFOs and the values of the auxiliary input bits. See the bit definitions in the IpBiS_MlrDef.h header file for more information.

IOCTL_IPBIS_MLR_GET_STATUS1

Function: Returns the interrupt/error status.
Input: none
Output: USHORT
Notes: Returns Status information for a given board obtained from the IPBIS_MLR_STATUS_1 register. This consists of latched error and interrupt status bits indicating the cause of an error or interrupt. After the status is read, a value is written back to this register to clear only the specific latched bits that were read. This will insure that no interrupt cause is missed due to being asserted between the read and write cycles. Note: The RX_MISSD_CNT error bit is not cleared in this manner, but by reading the Rx word count register. See the bit definitions in the IpBiS_MlrDef.h header file for more information.

IOCTL_IPBIS_MLR_GET_RX_WRDCNT

Function: Returns the number of received words.
Input: none
Output: USHORT
Notes: Returns the number of words received in the last data reception. If a new value is written i.e. another reception occurs before the old count is read, the RX_MISSD_CNT error bit is set. Reading the word count clears the error bit.

IOCTL_IPBIS_MLR_SET_IP_CONTROL

Function: Sets the configuration of the board slot.
Input: ULONG
Output: none
Notes: Controls the IP clock speed and interrupt enables for the IP slot that the board occupies. See the bit definitions in the IpBiS_MlrDef.h header file for more information.

IOCTL_IPBIS_MLR_GET_IP_CONTROL

Function: Returns the configuration of the board slot.
Input: none
Output: ULONG
Notes: Returns the slot configuration register value. See the bit definitions in the IpBiS_MlrDef.h header file for more information.

IOCTL_IPBIS_MLR_SET_TX_CONFIG

Function: Sets the transmitter configuration bits.
Input: USHORT
Output: none
Notes: Controls the Tx and Tx FIFO almost empty interrupt enables and the auxiliary output bit values and direction controls. See the bit definitions in the IpBiS_MlrDef.h header file for more information.

IOCTL_IPBIS_MLR_GET_TX_CONFIG

Function: Returns the transmitter configuration bits.
Input: none
Output: USHORT
Notes: Returns the state of the bits listed above as well as the Tx start bit.

IOCTL_IPBIS_MLR_SET_RX_CONFIG

Function: Sets the receiver configuration bits.
Input: USHORT
Output: none
Notes: Controls the Rx and Rx FIFO almost full interrupt enables, the Rx auto-clear enable, the sync pattern init-only control bit, and the Rx RS-422/TTL input select.

IOCTL_IPBIS_MLR_GET_RX_CONFIG

Function: Returns the receiver configuration bits.
Input: none
Output: USHORT
Notes: Returns the state of the bits listed above as well as the Rx start bit.

IOCTL_IPBIS_MLR_SET_FRAME_SIZE

Function: Sets the number of 32-bit words in a data frame (defaults to 64).
Input: USHORT
Output: none
Notes: Controls the number of words in a data frame. The receiver expects to see the sync pattern as the first word of each frame.

IOCTL_IPBIS_MLR_GET_FRAME_SIZE

Function: Returns the data frame size.
Input: none
Output: USHORT
Notes: If no value is entered, the frame size defaults to 64 32-bit words.

IOCTL_IPBIS_MLR_SET_CLOCK_CONFIG

Function: Sets the clock configuration.

Input: USHORT
Output: none
Notes: Controls the clock divisor and whether the input clock or the divided clock is put out.

IOCTL_IPBIS_MLR_GET_CLOCK_CONFIG

Function: Returns the clock configuration.
Input: none
Output: USHORT
Notes: Returns the clock divisor and the output clock select control bits.

IOCTL_IPBIS_MLR_SET_FIFO_LEVELS

Function: Sets the Tx FIFO almost empty and Rx FIFO almost full levels.
Input: FIFO_LEVELS struct
Output: none
Notes: Sets the almost full level for the Rx FIFO; the value is the number of words below full that the PAF flag becomes asserted. Sets the almost empty level for the Tx FIFO; the value is the number of words above empty for which the PAE flag is asserted. The Tx and Rx 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_IPBIS_MLR_GET_FIFO_LEVELS

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

IOCTL_IPBIS_MLR_RESET_FIFOS

Function: Resets the transmit and receive FIFOs.
Input: none
Output: none

Notes: Resets both FIFOs. This will clear all data and reset the almost full and empty values to their default values.

IOCTL_IPBIS_MLR_START_TX

Function: Starts the Tx state machine.
Input: none
Output: none
Notes: Provided data has been loaded into the FIFO, this command will send transmit data.

IOCTL_IPBIS_MLR_STOP_TX

Function: Stops the Tx state machine.
Input: none
Output: none
Notes: This command will abort a transmission. Used mainly to stop the transmitter if no data is loaded

IOCTL_IPBIS_MLR_START_RX

Function: Starts the Rx state machine.
Input: none
Output: none
Notes: This command will enable the receiver to start looking for data.

IOCTL_IPBIS_MLR_STOP_RX

Function: Stops the Rx state machine.
Input: none
Output: none
Notes: This command will abort a reception. Used mainly to stop the Rx state machine when it is waiting for data, but no data is being received.

IOCTL_IPBIS_MLR_REGISTER_EVENT

Function: Registers an event to be signaled when an interrupt occurs.
Input: Handle to Event object
Output: none
Notes: The caller creates an event with CreateEvent() and supplies the handle returned from that call as the input to this IOCTL. The driver then obtains a system pointer to the event and signals the event when an interrupt is serviced. The user interrupt service routine waits on this event, allowing it to respond to the interrupt. In order to unregister the event, set the event handle to NULL while making this call.

IOCTL_IPBIS_MLR_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 control2 register the same. Also checks the state of the IP slot control register interrupt 0 enable bit in the saved configuration, and sets it if needed. 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_IPBIS_MLR_DISABLE_INTERRUPT

Function: Sets the master interrupt enable to FALSE.
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_IPBIS_MLR_FORCE_INTERRUPT

Function: Causes a system interrupt to occur.
Input: none
Output: none
Notes: Causes an interrupt to be asserted on the IP bus if the master interrupt enable is set. This IOCTL is used for development, to test interrupt processing.

IOCTL_IPBIS_MLR_SET_VECTOR

Function: Sets the value of the interrupt vector.
Input: UCHAR
Output: none
Notes: This value is driven onto the low byte of the data bus in response to an INT_SEL strobe, which is used in vectored interrupt cycles. This value will be read in the interrupt service routine and stored for future reference.

IOCTL_IPBIS_MLR_GET_VECTOR

Function: Returns the current interrupt vector value.
Input: none
Output: UCHAR
Notes:

IOCTL_IPBIS_MLR_SET_SYNC_PATTERN

Function: Sets the sync pattern register values.
Input: ULONG
Output: none
Notes: This value is used to synchronize the start of each receive data frame. If no value is entered, the sync pattern defaults to 0x7af33402.

IOCTL_IPBIS_MLR_GET_ SYNC_PATTERN

Function: Returns the value in the sync pattern registers.
Input: none
Output: ULONG
Notes:

IOCTL_IPBIS_MLR_GET_INT_STATUS

Function: Returns the interrupt status and interrupt vector.
Input: none
Output: INT_STAT struct
Notes: Returns the interrupt vector and the contents of the interrupt status register that were read in the last ISR call. These values are returned in the INT_STAT structure. See the bit definitions in the IpBiS_MlrDef.h header file and the struct definition in the DDIpBiS_Mlr.h header file for more information.

Write

Data to be sent from the transmitter is written to the transmit FIFO using a WriteFile() call. The user supplies the device handle, a pointer to the buffer containing the data, the number of bytes to write, a pointer to a variable to store the amount of data actually transferred, and a pointer to an optional Overlapped structure for performing asynchronous IO. The number of bytes is checked to see if it exceeds the size of the FIFO and if not the command is executed with successive writes to the Tx FIFO port. The driver takes advantage of the carrier 32-bit double-write capability to load two FIFO words with a single PCI write until less than four bytes remain in the buffer. See Win32 help files for details of the WriteFile() call.

Read

Received data is read from the receive FIFO using a ReadFile() call. The user supplies the device handle, a pointer to the buffer in which to store the data, the number of bytes to read, a pointer to a variable to store the

amount of data actually transferred, and a pointer to an optional Overlapped structure for performing asynchronous IO. The number of bytes is checked to see if it exceeds the size of the FIFO and if not, the command is executed with successive reads from the Rx FIFO port. The driver takes advantage of the carrier 32-bit double-read capability to read two FIFO words with a single PCI read until less than four bytes remain to be read. See Win32 help files for the details of the ReadFile() call.

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