

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

PcBa14 & Ba14Chan

Driver Documentation

Win32 Driver Model

Revision A

Corresponding Hardware: Revision A

10-2006-0801

Corresponding Firmware: Revision A

PcBa14, Ba14Chan
WDM Device Drivers for the
PC104pBiSerial-III
1-Channel Serial Interface

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

©2006 by Dynamic Engineering.
Other trademarks and registered trademarks are owned by their
respective manufactures.
Manual Revision A. Revised July 26, 2006

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

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

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

Dynamic Engineering's products are not authorized for use as critical components in life support devices or systems without the express written approval of the president of Dynamic Engineering.

Connection of incompatible hardware is likely to cause serious damage.



Table of Contents

Introduction	5
Note	5
Driver Installation	5
Windows 2000 Installation	6
Windows XP Installation	6
Driver Startup	7
IO Controls	13
IOCTL_PC_BA14_GET_INFO	13
IOCTL_PC_BA14_SET_CONFIG	13
IOCTL_PC_BA14_GET_CONFIG	13
IOCTL_PC_BA14_GET_STATUS	13
IOCTL_PC_BA14_SET_DIR_TERM	14
IOCTL_PC_BA14_GET_DIR_TERM	14
IOCTL_PC_BA14_SET_IO_CONFIG	14
IOCTL_PC_BA14_GET_IO_CONFIG	14
IOCTL_PC_BA14_READ_IO_DATA	14
IOCTL_PC_BA14_SET_TTL_CONFIG	15
IOCTL_PC_BA14_GET_TTL_CONFIG	15
IOCTL_PC_BA14_READ_TTL_DATA	15
IOCTL_PC_BA14_REGISTER_EVENT	15
IOCTL_PC_BA14_ENABLE_INTERRUPT	15
IOCTL_PC_BA14_DISABLE_INTERRUPT	16
IOCTL_PC_BA14_FORCE_INTERRUPT	16
IOCTL_PC_BA14_GET_ISR_STATUS	16
IOCTL_BA14_CHAN_GET_INFO	16
IOCTL_BA14_CHAN_RESET_FIFOS	16
IOCTL_BA14_CHAN_SET_CONFIG	17
IOCTL_BA14_CHAN_GET_CONFIG	17
IOCTL_BA14_CHAN_GET_STATUS	17
IOCTL_BA14_CHAN_SET_FIFO_LEVELS	17
IOCTL_BA14_CHAN_GET_FIFO_LEVELS	17
IOCTL_BA14_CHAN_WRITE_FIFO	18
IOCTL_BA14_CHAN_READ_FIFO	18
IOCTL_BA14_CHAN_GET_FIFO_COUNTS	18
IOCTL_BA14_CHAN_REGISTER_EVENT	18
IOCTL_BA14_CHAN_ENABLE_INTERRUPT	18
IOCTL_BA14_CHAN_DISABLE_INTERRUPT	19
IOCTL_BA14_CHAN_FORCE_INTERRUPT	19
IOCTL_BA14_CHAN_GET_ISR_STATUS	19
Write	20
Read	20
Warranty and Repair	20
Service Policy	21



Out of Warranty Repairs	21
For Service Contact:	21



Introduction

The PcBa14 and Ba14Chan drivers are Win32 driver model (WDM) device drivers for the PC104p-BiSerial-III from Dynamic Engineering. The PC104p-BiSerial-III board has a Spartan3-1500 Xilinx FPGA to implement the PCI interface, FIFOs and protocol control and status for one serial channel. The channel has two 2k x 32-bit data FIFOs for data transmission and reception.

When the PC104p-BiSerial-III is recognized by the PCI bus configuration utility it will start the PcBa14 driver. The PcBa14 driver enumerates the channel and creates a separate Ba14Chan device object. This allows the I/O channel 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 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 PC104p-BiSerial-III user manual (also referred to as the hardware manual).

Driver Installation

There are several files provided in each driver package. These files include PcBa14.sys, PcBa14.inf, DDPcBa14.h, PcBa14GUID.h, Ba14Chan.sys, Ba14Chan.inf, DDBa14Chan.h, Ba14ChanGUID.h, PcBa14Test.exe, and PcBa14Test source files.



Windows 2000 Installation

Copy PcBa14.inf, Ba14Chan.inf, PcBa14.sys and Ba14Chan.sys to a floppy disk, or CD if preferred.

With the PC104p-BiSerial-III 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 PcBa14.inf file.
- Select *Next*.
- Select *Finish* to close the *Found New Hardware Wizard*.

The system should now see the PcBa14 channels and reopen the *New Hardware Wizard*. Proceed as above substituting Ba14Chan.inf for PcBa14.inf.

Windows XP Installation

Copy PcBa14.inf, Ba14Chan.inf, PcBa14.sys and Ba14Chan.sys to a floppy disk, or CD if preferred.

With the PC104p-BiSerial-III hardware installed, power-on the PCI host computer and wait for the *Found New Hardware Wizard* dialogue window to appear.

- Insert the disk prepared above in the desired drive.
- Select *No when asked to connect to Windows Update*.
- Select *Next*.
- Select *Install the software automatically*.
- Select *Next*.
- Select *Finish* to close the *Found New Hardware Wizard*.

The system should now see the PcBa14 channel and reopen the *New Hardware Wizard*.



The DDPcBa14.h and DDBa14Chan.h files are C header files that define the Application Program Interface (API) to the drivers. The PcBa14GUID.h and Ba14ChanGUID.h files are C header files that define the device interface identifiers for the PcBa14 and Ba14Chan 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 PcBa14Test.exe file is a sample Win32 console application that makes calls into the PcBa14/Ba14Chan drivers to test each driver call without actually writing any application code. It is not required during the driver installation.

To run PcBa14Test.exe, open a command prompt console window and type *PcBa14Test -dO -?* to display a list of commands (the PcBa14Test.exe file must be in the directory that the window is referencing). The commands are all of the form *PcBa14Test -dn -im* where *n* and *m* are the device number and driver PcBa14 ioctl number respectively or *PcBa14Test -cn -im* where *n* and *m* are the channel number and Ba14Chan driver ioctl number respectively. This application is intended to test the proper functioning of the driver calls, not for normal operation.

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 PcBa14GUID.h and Ba14ChanGUID.h.

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



```

// The maximum length of the device name for a given interface
#define MAX_DEVICE_NAME 256

// Handles to the device objects
HANDLE hPcBa14 = INVALID_HANDLE_VALUE;
HANDLE hBa14Chan[PC_BA14_NUM_CHANNELS] = {INVALID_HANDLE_VALUE};

// Pc104-Biserial-III channel number
UCHAR chan;

// 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_PC_BA14,
    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 devNum
if(!SetupDiEnumDeviceInterfaces(hDeviceInfo,
    NULL,
    (LPGUID)&GUID_DEVINTERFACE_PC_BA14,
    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);
}
}

// 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
lstrcpy(deviceName,
        pDeviceDetail->DevicePath,
        MAX_DEVICE_NAME);

// Cleanup search
free(pDeviceDetail);

```



```

SetupDiDestroyDeviceInfoList(hDeviceInfo);

// Open driver
// Create the handle to the device
hPcBa14 = CreateFile(deviceName,
                    GENERIC_READ | GENERIC_WRITE,
                    FILE_SHARE_READ | FILE_SHARE_WRITE,
                    NULL,
                    OPEN_EXISTING,
                    NULL,
                    NULL);

if(hPcBa14 == INVALID_HANDLE_VALUE)
{
    printf("***Error: couldn't open %s, (%d)\n",
           deviceName,
           GetLastError());

    exit(-1);
}

hDeviceInfo = SetupDiGetClassDevs(
                (LPGUID)&GUID_DEVINTERFACE_BA14_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 = devNum * PC_BA14_NUM_CHANNELS;
     chan < (devNum + 1) * PC_BA14_NUM_CHANNELS;
     chan++)
{
    // Find the interface for chan devices
    if(!SetupDiEnumDeviceInterfaces(
        hDeviceInfo,
        NULL,
        (LPGUID)&GUID_DEVINTERFACE_BA14_CHAN,
        chan,
        &interfaceData))
    {
        status = GetLastError();
        if(status == ERROR_NO_MORE_ITEMS)
        {
            printf("***Error: couldn't find device(no more items), (%d)\n",
                   chan);

            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);

// Open driver
// Create the handle to the device
hBa14Chan[chan] = CreateFile(deviceName,
                             GENERIC_READ   | GENERIC_WRITE,
                             FILE_SHARE_READ | FILE_SHARE_WRITE,
                             NULL,
                             OPEN_EXISTING,
                             NULL,
                             NULL);

if(hBa14Chan[chan] == INVALID_HANDLE_VALUE)
{
    printf("***Error: couldn't open %s, (%d)\n",
           deviceName,
           GetLastError());

    exit(-1);
}
}

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 these drivers are as follows:

IOCTL_PC_BA14_GET_INFO

Function: Returns the Driver version, Xilinx flash revision, Switch value, and Instance number.

Input: None

Output: PC_BA14_DRIVER_DEVICE_INFO structure

Notes: Switch value is the configuration of the 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 DDPcBa14.h for the definition of PC_BA14_DRIVER_DEVICE_INFO.

IOCTL_PC_BA14_SET_CONFIG

Function: Writes to the base configuration register on the PC104p-BiSerial-III.

Input: Value of control register (unsigned long integer)

Output: None

Notes: Only the bits in the BASE_CONFIG_MASK are controlled by this command. See the bit definitions in PcBa14Def.h for information on determining this value.

IOCTL_PC_BA14_GET_CONFIG

Function: Returns the configuration of the base control register.

Input: None

Output: Value of control register (unsigned long integer)

Notes: The return value includes the bits in BASE_CONFIG_MASK and BASE_MASTER_INT_EN. This command is used mainly for testing.

IOCTL_PC_BA14_GET_STATUS

Function: Returns the value of the status register.

Input: None

Output: Value of status register (unsigned long integer)

Notes: See DDPcBa14.h for the status bit definitions.



IOCTL_PC_BA14_SET_DIR_TERM

Function: Sets the direction (input or output) and termination (off or on) of the 16 RS-485 I/O lines.

Input: PC_BA14_DIR_TERM structure

Output: None

Notes: The bits in each of the structure fields operate on the respective I/O line i.e. if direction bit 0 is a one, I/O line 0 is an output; if termination bit 6 is a one, I/O line 6 is terminated etc. See DDPcBa14.h for the definition of PC_BA14_DIR_TERM.

IOCTL_PC_BA14_GET_DIR_TERM

Function: Returns the direction and termination of the 16 RS-485 I/O lines.

Input: None

Output: PC_BA14_DIR_TERM structure

Notes: See DDPcBa14.h for the definition of PC_BA14_DIR_TERM.

IOCTL_PC_BA14_SET_IO_CONFIG

Function: Sets the source and data value of the 16 RS-485 output lines.

Input: PC_BA14_485_DATA_CNTL structure

Output: None

Notes: The bits in each of the structure fields operate on the respective I/O line to specify the data when that line is configured as an output. When a bit in the Select field is a one, the data source for the I/O line is the register loaded from the Data field. Otherwise the data source is the transmit I/O state machine. See DDPcBa14.h for the definition of PC_BA14_485_DATA_CNTL.

IOCTL_PC_BA14_GET_IO_CONFIG

Function: Returns the source and data value of the 16 RS-485 output lines.

Input: None

Output: PC_BA14_485_DATA_CNTL structure

Notes: See DDPcBa14.h for the definition of PC_BA14_485_DATA_CNTL.

IOCTL_PC_BA14_READ_IO_DATA

Function: Returns the data values on the 16 RS-485 input lines.

Input: None

Output: Unsigned short integer.

Notes:



IOCTL_PC_BA14_SET_TTL_CONFIG

Function: Sets the enables and data value of the 8 TTL output lines.

Input: PC_BA14_TTL_DATA_CNTL structure

Output: None

Notes: When a bit in the Enable field is a one, the value of the respective bit in the Data field is driven onto the respective TTL line. See DDPcBa14.h for the definition of PC_BA14_TTL_DATA_CNTL.

IOCTL_PC_BA14_GET_TTL_CONFIG

Function: Returns the enables and data value of the 8 TTL output lines.

Input: None

Output: PC_BA14_TTL_DATA_CNTL structure

Notes: See DDPcBa14.h for the definition of PC_BA14_TTL_DATA_CNTL.

IOCTL_PC_BA14_READ_TTL_DATA

Function: Returns the data values on the 8 TTL input lines.

Input: None

Output: Unsigned character.

Notes:

IOCTL_PC_BA14_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.

IOCTL_PC_BA14_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 it.



IOCTL_PC_BA14_DISABLE_INTERRUPT

Function: Disables the master interrupt.

Input: none

Output: none

Notes: This call is used when local interrupt processing is no longer desired.

IOCTL_PC_BA14_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 master interrupt is enabled. This IOCTL is used for development, to test interrupt processing.

IOCTL_PC_BA14_GET_ISR_STATUS

Function: Returns the interrupt status that was 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 serviced.

IOCTL_BA14_CHAN_GET_INFO

Function: Returns the Driver version and Instance number.

Input: None

Output: BA14_CHAN_DRIVER_DEVICE_INFO structure

Notes: See DDPcBa14.h for the definition of BA14_CHAN_DRIVER_DEVICE_INFO.

IOCTL_BA14_CHAN_RESET_FIFOS

Function: Resets the channel's FIFOs.

Input: None

Output: None

Notes: Resets the Tx and Rx FIFOs for the referenced channel.



IOCTL_BA14_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 DDBa14Chan.h for information on determining this value.

IOCTL_BA14_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 and CNTRL_DMA_WREN, CNTRL_DMA_RDEN and CNTRL_MINTEN. This command is used mainly for testing.

IOCTL_BA14_CHAN_GET_STATUS

Function: Returns the channel's status value.

Input: None

Output: Value of the channel's status register (unsigned long integer)

Notes: See DDBa14Chan.h for the status bit definitions.

IOCTL_BA14_CHAN_SET_FIFO_LEVELS

Function: Sets the channel's receiver almost full and transmitter empty levels.

Input: BA14_CHAN_FIFO_LEVELS structure

Output: None

Notes: The FIFO levels are used to determine status when the FIFO data counts reach the specified levels. See DDBa14Chan.h for the definition of BA14_CHAN_FIFO_LEVELS.

IOCTL_BA14_CHAN_GET_FIFO_LEVELS

Function: Returns the channel's receiver almost full and transmitter empty levels.

Input: None

Output: BA14_CHAN_FIFO_LEVELS structure

Notes: See DDBa14Chan.h for the definition of BA14_CHAN_FIFO_LEVELS.



IOCTL_BA14_CHAN_WRITE_FIFO

Function: Writes a data word the channel's transmit FIFO.

Input: FIFO data word (unsigned long integer)

Output: None

Notes:

IOCTL_BA14_CHAN_READ_FIFO

Function: Reads a data word from the channel's receive FIFO.

Input: None

Output: FIFO data word (unsigned long integer)

Notes:

IOCTL_BA14_CHAN_GET_FIFO_COUNTS

Function: Returns the number of data words in the transmit and receive FIFOs.

Input: None

Output: BA14_CHAN_FIFO_COUNTS structure

Notes: See DDBa14Chan.h for the definition of BA14_CHAN_FIFO_COUNTS.

IOCTL_BA14_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_BA14_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 it.



IOCTL_BA14_CHAN_DISABLE_INTERRUPT

Function: Disables the master interrupt.

Input: none

Output: none

Notes: This call is used when local interrupt processing is no longer desired.

IOCTL_BA14_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 master interrupt is enabled. This IOCTL is used for development, to test interrupt processing.

IOCTL_BA14_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 interrupt serviced.



Write

PC104p-BiSerial-III 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 IO.

Read

PC104p-BiSerial-III 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 IO.

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

All information provided is Copyright Dynamic Engineering.

