

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

DdlBase & DdlChan

Driver Documentation

Win32 Driver Model

Revision B

Corresponding Hardware: Revision A

10-2006-0801

Corresponding Firmware: Revision B

DdlBase & DdlChan
WDM Device Drivers for the
PC104pBiSerial-III
Digital Data-Link Serial Interface

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

©2006 by Dynamic Engineering.
Other trademarks and registered trademarks are
owned by their respective manufactures.
Manual Revision B. Revised November 28, 2007

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	4
Note	4
Driver Installation.....	4
Windows 2000 Installation	5
Windows XP Installation	5
Driver Startup	6
IO Controls.....	11
IOCTL_DDL_BASE_GET_INFO	11
IOCTL_DDL_BASE_LOAD_PLL_DATA.....	11
IOCTL_DDL_BASE_READ_PLL_DATA.....	11
IOCTL_DDL_CHAN_GET_INFO.....	11
IOCTL_DDL_CHAN_SET_CONFIG.....	12
IOCTL_DDL_CHAN_GET_STATE	12
IOCTL_DDL_CHAN_GET_STATUS.....	12
IOCTL_DDL_CHAN_SET_FIFO_LEVELS	12
IOCTL_DDL_CHAN_GET_FIFO_LEVELS.....	12
IOCTL_DDL_CHAN_GET_FIFO_COUNTS.....	12
IOCTL_DDL_CHAN_RESET_FIFOS	13
IOCTL_DDL_CHAN_WRITE_FIFO.....	13
IOCTL_DDL_CHAN_READ_FIFO	13
IOCTL_DDL_CHAN_SET_TX_CONFIG.....	13
IOCTL_DDL_CHAN_GET_TX_STATE.....	14
IOCTL_DDL_CHAN_SET_RX_CONFIG	14
IOCTL_DDL_CHAN_GET_RX_STATE	14
IOCTL_DDL_CHAN_START_TX.....	14
IOCTL_DDL_CHAN_STOP_TX	15
IOCTL_DDL_CHAN_START_RX.....	15
IOCTL_DDL_CHAN_STOP_RX.....	15
IOCTL_DDL_CHAN_REGISTER_EVENT.....	15
IOCTL_DDL_CHAN_ENABLE_INTERRUPT.....	16
IOCTL_DDL_CHAN_DISABLE_INTERRUPT.....	16
IOCTL_DDL_CHAN_FORCE_INTERRUPT.....	16
IOCTL_DDL_CHAN_GET_ISR_STATUS.....	16
Write	17
Read.....	17
Warranty and Repair.....	18
Service Policy.....	18
Out of Warranty Repairs.....	18
For Service Contact:.....	18

Introduction

The DdlBase and DdlChan drivers are Win32 driver model (WDM) device drivers for the PC104p-BiSerial-III-DDL from Dynamic Engineering. The PC104p-BiSerial-III-DDL has a Spartan3-1500 Xilinx FPGA to implement the PCI interface, four 2k x 32-bit internal data FIFOs, protocol control and status for two bidirectional digital data-link channels. The board also has an onboard programmable PLL to provide a timing reference and eight bi-directional RS-485 drivers for the external serial interface (transmit and receive data, clock and enable).

When the PC104p-BiSerial-III-DDL is recognized by the PCI bus configuration utility it will start the DdlBase driver. The DdlBase driver enumerates the two channels and creates separate DdlChan 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 device using bus-master DMA.

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-DDL user manual (also referred to as the hardware manual).

Driver Installation

There are several files provided the driver package. These files include DdlBase.sys, DdlChan.sys, Pc104pDdl.inf, DDDdlBase.h, DDDdlChan.h, DdlBaseGUID.h, DdlChanGUID.h, DdlTest.exe, and DdlTest source files.

The DDDdlBase.h and DDDdlChan.h files are C header files that defines the Application Program Interface (API) to the drivers. The DdlBaseGUID.h, DdlChanGUID.h files are C header files that define the device interface identifier for the 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. DdlBase.sys and DdlChan.sys are the actual driver executables that are loaded into kernel memory to provide the software/hardware interface to control the PC104p-BiSerial-III-DDL. Pc104pDdl.inf is an information file to allow the system to identify the proper drivers and where to load them. It also specifies registry information to be entered to facilitate plug-and-play functionality for the PC104p-BiSerial-III-DDL.

The DdlTest.exe file is a sample Win32 console application that makes calls into the DdlBase and DdlChan drivers to test each driver call without actually writing any application code. It is also not required for driver installation.



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

Windows 2000 Installation

Copy Pc104pDdl.inf, DdlBase.sys and DdlChan.sys to a floppy disk, or CD if preferred.

With the PC104p-BiSerial-III hardware installed, power-on the 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 and identify the Pc104pDdl.inf file.
- Select **Next**.
- Select **Finish** to close the **Found New Hardware Wizard**.

The system should now see the Pc104pDdl I/O channels and reopen the **New Hardware Wizard**. Proceed as above for each channel as required.

Windows XP Installation

Copy Pc104pDdl.inf, DdlBase.sys and DdlChan.sys to a floppy disk, or CD if preferred.

With the PC104p-BiSerial-III hardware installed, power-on the 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 Pc104pDdl I/O channels and reopen the **New Hardware Wizard**. Proceed as above for each channel as required.

Driver Startup

Once the driver has been installed it will start automatically when the system recognizes the hardware.

Handles can be opened to a specific board by using the CreateFile() function call and passing in the device name obtained from the system.

The interfaces to the device and I/O channels are identified using globally unique identifiers (GUIDs), which are defined in DdlBaseGUID.h and DdlChanGUID.h.

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

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

// Handles to device objects
HANDLE hDdlBase = INVALID_HANDLE_VALUE;
HANDLE hDdlChan[DDL_BASE_NUM_CHANNELS] = {INVALID_HANDLE_VALUE,
                                           INVALID_HANDLE_VALUE};

// Pc104p-DDL device number
ULONG devNum;

// Ddl I/O channel handle array index and instance number
ULONG chan, chanDev;

// Return status from command
LONG status;

// Handle to device 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 the devices
SP_DEVICE_INTERFACE_DATA interfaceData;
PSP_DEVICE_INTERFACE_DETAIL_DATA pDeviceDetail;

hDeviceInfo = SetupDiGetClassDevs(
    (LPGUID)&GUID_DEVINTERFACE_DDL_BASE,
    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_DDL_BASE,
                                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
lstrcpy(deviceName,
        pDeviceDetail->DevicePath,
        MAX_DEVICE_NAME);

// Cleanup search
free(pDeviceDetail);
SetupDiDestroyDeviceInfoList(hDeviceInfo);

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

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

// Now get channel info
hDeviceInfo = SetupDiGetClassDevs(
    (LPGUID)&GUID_DEVINTERFACE_DDL_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 * DDL_BASE_NUM_CHANNELS;
    chan < DDL_BASE_NUM_CHANNELS;
    chan++, chanDev++)
{
    // Find the interface for chanDev device
    if(!SetupDiEnumDeviceInterfaces(hDeviceInfo,
                                    NULL,
                                    (LPGUID)&GUID_DEVINTERFACE_DDL_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
hDdlChan[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(hDdlChan[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. 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 this driver are as follows:

IOCTL_DDL_BASE_GET_INFO

Function: Returns the Driver version, Xilinx design revision, PLL device ID, User Switch value, and Instance number.

Input: None

Output: DDL_BASE_DRIVER_DEVICE_INFO structure

Notes: The PLL device ID is used by the driver to communicate with the onboard PLL over its I2C serial bus, Switch value is the current setting 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 DDDdlBase.h for the definition of DDL_BASE_DRIVER_DEVICE_INFO.

IOCTL_DDL_BASE_LOAD_PLL_DATA

Function: Loads the internal registers of the PLL.

Input: DDL_BASE_PLL_DATA structure

Output: None

Notes: The DDL_BASE_PLL_DATA structure has only one field: Data – an array of 40 bytes containing the PLL register data to write to the PLL device. During the driver initialization, the PLL is loaded with default data that sets the 8x clock to 800 kHz to provide a data-rate of 100k bits/second. This call is only needed if some other data-rate is desired.

IOCTL_DDL_BASE_READ_PLL_DATA

Function: Returns the contents of the PLL's internal registers

Input: None

Output: DDL_BASE_PLL_DATA structure

Notes: The register data is output in the DDL_BASE_PLL_DATA structure in an array of 40 bytes.

IOCTL_DDL_CHAN_GET_INFO

Function: Returns the Driver version and Instance number.

Input: None

Output: DDL_CHAN_DRIVER_DEVICE_INFO structure

Notes: Instance number is the zero-based device number. See DDDdlChan.h for the definition of DDL_CHAN_DRIVER_DEVICE_INFO.



IOCTL_DDL_CHAN_SET_CONFIG

Function: Configures the channel control register for an I/O channel on the PC104p-BiSerial-III-DDL.

Input: DDL_CHAN_CONFIG structure

Output: None

Notes: Sets the configuration parameters for the channel interface, including interrupt enables master/slave setting and termination enable. See DDDdlChan.h for the definition of DDL_CHAN_CONFIG. Also see the hardware manual for detailed descriptions of the control parameter functions.

IOCTL_DDL_CHAN_GET_STATE

Function: Returns the configuration of the channel control register.

Input: None

Output: DDL_CHAN_STATE structure

Notes: Returns the fields set in the previous call as well as the states of the master interrupt enable and the read and write DMA enables. This command is used mainly for testing.

IOCTL_DDL_CHAN_GET_STATUS

Function: Returns the value of the channel status register and clears the latched status bits.

Input: None

Output: Value of status register (unsigned long integer)

Notes: See DDDdlChan.h for the status bit definitions. The bits in STAT_LATCH_MASK will be cleared if and only if they are set when the status is read.

IOCTL_DDL_CHAN_SET_FIFO_LEVELS

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

Input: DDL_CHAN_FIFO_LEVELS structure

Output: None

Notes: The FIFO levels are used to determine the transmit FIFO almost empty and receive FIFO almost full status. These status bits will change state when the FIFO data counts reach the specified levels. See DDDdlChan.h for the definition of DDL_CHAN_FIFO_LEVELS.

IOCTL_DDL_CHAN_GET_FIFO_LEVELS

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

Input: None

Output: DDL_CHAN_FIFO_LEVELS structure

Notes: See DDDdlChan.h for the definition of DDL_CHAN_FIFO_LEVELS.



IOCTL_DDL_CHAN_GET_FIFO_COUNTS

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

Input: None

Output: DDL_CHAN_FIFO_COUNTS structure

Notes: The counts returned include data in the transmit and receive data pipelines. The transmit FIFO maximum count is 0x801 (2049) and the receive FIFO maximum count is 0x804 (2052). See DDDdlChan.h for the definition of DDL_CHAN_FIFO_COUNTS.

IOCTL_DDL_CHAN_RESET_FIFOS

Function: Resets the channel's FIFOs.

Input: DDL_CHAN_FIFO_SEL enumerated type.

Output: None

Notes: Resets either the transmit, receive or both FIFOs depending on the value of the input parameter. See DDDdlChan.h for the definition of DDL_CHAN_FIFO_SEL.

IOCTL_DDL_CHAN_WRITE_FIFO

Function: Writes one 32-bit data word to the channel's transmit FIFO.

Input: FIFO data word (unsigned long integer)

Output: None

Notes: Useful for small data transfers. Use the WriteFile command for much better performance for larger transfers.

IOCTL_DDL_CHAN_READ_FIFO

Function: Reads one 32-bit data word from the channel's receive FIFO.

Input: None

Output: FIFO data word (unsigned long integer)

Notes: Useful for small data transfers. Use the ReadFile command for much better performance for larger transfers.

IOCTL_DDL_CHAN_SET_TX_CONFIG

Function: Writes to the channel transmitter's configuration register.

Input: DDL_CHAN_TX_CONFIG structure.

Output: None

Notes: Sets the configuration parameters for the transmit interface. See DDDdlChan.h for the definition of DDL_CHAN_TX_CONFIG. Also see the hardware manual for detailed descriptions of the control parameter functions.



IOCTL_DDL_CHAN_GET_TX_STATE

Function: Returns the configuration of the channel transmit control register.

Input: None

Output: DDL_CHAN_TX_STATE structure.

Notes: Returns the values written in the previous call as well as the state of the transmitter start latch.

IOCTL_DDL_CHAN_SET_RX_CONFIG

Function: Writes to the channel receiver's configuration register.

Input: DDL_CHAN_RX_CONFIG structure.

Output: None

Notes: Sets the configuration parameters for the receive interface. See DDDdlChan.h for the definition of DDL_CHAN_RX_CONFIG. Also see the hardware manual for detailed descriptions of the control parameter functions.

IOCTL_DDL_CHAN_GET_RX_STATE

Function: Returns the configuration of the channel receiver's control register.

Input: None

Output: DDL_CHAN_RX_STATE structure.

Notes: Returns the values written in the previous call as well as the state of the receiver start latch.

IOCTL_DDL_CHAN_START_TX

Function: Starts the channel transmit interface.

Input: None

Output: None

Notes: This command must be run to start a data transmission. If the channel is in master mode, serial data transmission will commence when this call is made provided that there is data in the transmit FIFO. If data is not present the transmitter will wait for data to be written before the transmission begins. When the FIFO becomes empty, an interrupt will occur provide the appropriate enables have been set. If the ClearEnable field was true when the transmit configuration was set, the transmit enable will be automatically cleared when the FIFO becomes empty. If both the receiver and transmitter are enabled when the channel is in slave mode, the receiver has precedence and the transmitter will not start until the receive enable is cleared.

IOCTL_DDL_CHAN_STOP_TX

Function: Stops the channel transmitter.

Input: None

Output: None

Notes: This call is used to stop the transmitter. If the ClearEnable field was true when the transmit configuration was set, the transmit enable will be automatically cleared when the transmit FIFO becomes empty and this call will not be needed.

IOCTL_DDL_CHAN_START_RX

Function: Starts the channel receiver.

Input: None

Output: None

Notes: This command must be run to start a data reception. If the channel is in master mode, the serial data transfer will commence when this call is made. If the channel is in slave mode it will wait for the enable and clock from the remote transmitter to receive and store data. When the specified number of words have been received, an interrupt will occur, provided the appropriate enables have been set. If the ClearEnable field was true when the receiver configuration was set, the receive enable will be automatically cleared at this time. If both the receiver and transmitter are enabled when the channel is in master mode, the transmitter has precedence and the receiver will not start until the transmit enable is cleared.

IOCTL_DDL_CHAN_STOP_RX

Function: Stops the channel receiver.

Input: None

Output: None

Notes: This call is used to stop the receiver. If the ClearEnable field was true when the receive configuration was set, the receiver enable will be automatically cleared when the specified number of words have been received and this call will not be needed.

IOCTL_DDL_CHAN_REGISTER_EVENT

Function: Registers an event to be signaled when a channel 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_DDL_CHAN_ENABLE_INTERRUPT

Function: Enables the channel master interrupt enable.

Input: None

Output: None

Notes: This command must be run to allow the channel to respond to user interrupts. The master interrupt enable is disabled in the driver interrupt service routine; therefore this command must be run after each user interrupt occurs to re-enable it. The DMA interrupts are handled automatically by the driver and do not require any user involvement.

IOCTL_DDL_CHAN_DISABLE_INTERRUPT

Function: Disables the channel master interrupt enable.

Input: None

Output: None

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

IOCTL_DDL_CHAN_FORCE_INTERRUPT

Function: Causes a channel 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_DDL_CHAN_GET_ISR_STATUS

Function: Returns the interrupt status that was read in the ISR from the channel's 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. See DDDdlChan.h for the status bit definitions. The bits in STAT_LATCH_MASK will have been cleared in the ISR if and only if they were set when this status was read.

Write

PC104p-BiSerial-III-DDL 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-DDL 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
150 DuBois, Suite 3
Santa Cruz, CA 95060
(831) 457-8891 - Fax (831) 457-4793
support@dyneng.com

All information provided is Copyright Dynamic Engineering.

