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

RI1Base
&
RI1Chan

Driver Documentation

Win32 Driver Model

Revision A
Corresponding Hardware: Revision D
10-2005-0204
Corresponding Firmware: Revision A

Rl1Base, Rl1Chan
WDM Device Drivers for the
PMC-BiSerial-III-RL1
8-Channel PMC-Based UART Interface

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

Table of Contents

Introduction

The Rl1Base and Rl1Chan drivers are Win32 driver model (WDM) device drivers for the PMC-BiSerial-III RL1 from Dynamic Engineering. The PMC-BiSerial-III board has a Spartan3-1500 Xilinx FPGA to implement the PCI interface, FIFOs and protocol control and status for eight serial channels. Each channel has two 1k x 32-bit data FIFOs for data transmission and reception.

When the PMC-BiSerial-III RL1 is recognized by the PCI bus configuration utility it will start the Rl1Base driver. The Rl1Base driver enumerates the channels and creates eight separate Rl1Chan device objects. This allows the I/O channels to be totally independent while the base driver controls the device items that are common. IO Control calls (IOCTLs) are used to configure the board and read status. Read and Write calls are used to move blocks of data in and out of the I/O channel devices.

Note

This documentation will provide information about all calls made to the drivers, and how the drivers interact with the device for each of these calls. For more detailed information on the hardware implementation, refer to the PMC-BiSerial-III RL1 user manual (also referred to as the hardware manual).

Driver Installation

There are several files provided in each driver package. These files include PmcBis3Rl1.inf, Rl1Base.sys, DDRl1Base.h, Rl1BaseGUID.h, Rl1Chan.sys, DDRl1Chan.h, Rl1ChanGUID.h, Rl1Test.exe, and Rl1Test source files. DDRl1Base.h and DDRl1Chan.h are C header files that define the Application Program Interface (API) to the drivers. Rl1BaseGUID.h and Rl1ChanGUID.h are C header files that define the device interface identifiers for the Rl1Base and Rl1Chan 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.

Rl1Test.exe is a sample Win32 console application that makes calls into the Rl1Base/Rl1Chan drivers to test each driver call without actually writing any application code. It is not required during the driver installation.

To run Rl1Test.exe, open a command prompt console window and type a command. Type ***Rl1Test -d0 -?*** to display a list of commands (the Rl1Test.exe file must be in the directory that the window is referencing). The commands are all of the form ***Rl1Test -dn -im*** where ***n*** and ***m*** are the device number and driver Rl1Base ioctl number respectively or ***Rl1Test -cn -im*** where ***n*** and ***m*** are the channel number and Rl1Chan driver ioctl number respectively. This application is intended to test the proper functioning of the driver calls, not for normal operation.

Windows 2000 Installation

Copy PmcBis3Rl1.inf, Rl1Base.sys and Rl1Chan.sys to a floppy disk, or CD if preferred.

With the PMC-BiSerial-III RL1 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 Rl1Base.inf file.
- Select ***Next***.
- Select ***Finish*** to close the ***Found New Hardware Wizard***.

The system should now see the Rl1 I/O channels and reopen the ***New Hardware Wizard.*** Proceed as above for each channel as necessary.

Windows XP Installation

Copy PmcBis3Rl1.inf, Rl1Base.sys and Rl1Chan.sys to a floppy disk, or CD if preferred.

With the PMC-BiSerial-III RL1 hardware installed, power-on the PCI host computer and wait for the ***Found New Hardware Wizard*** dialogue window to appear.

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

The system should now see the Rl1 I/O channels and reopen the ***New Hardware Wizard***. Proceed as above for each channel as necessary.

Driver Startup

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

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

The interface to the device is identified using a globally unique identifier (GUID), which is defined in Rl1BaseGUID.h and Rl1ChanGUID.h.

Note: In order to build an application with the code below you must link with setupapi.lib.

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

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

// Handles to device objects
HANDLE hRl1Base                        =  INVALID_HANDLE_VALUE;
HANDLE hRl1Chan[RL1_BASE_NUM_CHANNELS] = {INVALID_HANDLE_VALUE,
                                          INVALID_HANDLE_VALUE,
                                          INVALID_HANDLE_VALUE,
                                          INVALID_HANDLE_VALUE,
                                          INVALID_HANDLE_VALUE,
                                          INVALID_HANDLE_VALUE,
                                          INVALID_HANDLE_VALUE,
                                          INVALID_HANDLE_VALUE};

// PMC-BiSerial-III Rl1 device number (starting with zero)
ULONG                            devNum;

// Rl1 channel handle array index and interface number
ULONG                            chan, i;

// Flag to indicate end of channel device search
BOOLEAN                          done = FALSE;

// Return length from driver call
ULONG                            length;

// Info structure to match proper channel instance number
RL1_CHAN_DRIVER_DEVICE_INFO      info;

// Return status from command
LONG                             status;

// Handle to device interface information structure
HDEVINFO                         hDeviceInfo;
```

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

// Size of buffer reguired to get the symbolic link name
DWORD                             requiredSize;

// Interface data structures for this device
SP_DEVICE_INTERFACE_DATA          interfaceData;
PSP_DEVICE_INTERFACE_DETAIL_DATA pDeviceDetail;

hDeviceInfo = SetupDiGetClassDevs(
                  (LPGUID)&GUID_DEVINTERFACE_RL1_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);

for(i = 0; i <= devNum; i++)
{// Find the interface for device devNum
   if(!SetupDiEnumDeviceInterfaces(hDeviceInfo,
                                   NULL,
                           (LPGUID)&GUID_DEVINTERFACE_RL1_BASE,
                                   i,
                                   &interfaceData))
   {
      status = GetLastError();
      if(status == ERROR_NO_MORE_ITEMS)
      {
         printf("**Error: couldn't find device(no more items), (%d)\n", i);
         SetupDiDestroyDeviceInfoList(hDeviceInfo);
         exit(-1);
      }
      else
      {
         printf("**Error: couldn't enum device, (%d)\n", status);
         SetupDiDestroyDeviceInfoList(hDeviceInfo);
         exit(-1);
      }
   }
}
```

```
// Found our device-get the details data to obtain the symbolic link name
if(!SetupDiGetDeviceInterfaceDetail(hDeviceInfo,
                                    &interfaceData,
                                    NULL,
                                    0,
                                    &requiredSize,
                                    NULL))
{
   if(GetLastError() != ERROR_INSUFFICIENT_BUFFER)
   {
      printf("**Error: couldn't get interface detail, (%d)\n",
              GetLastError());

      SetupDiDestroyDeviceInfoList(hDeviceInfo);
      exit(-1);
   }
}

// Allocate a buffer to get detail
pDeviceDetail = (PSP_DEVICE_INTERFACE_DETAIL_DATA)malloc(requiredSize);

if(pDeviceDetail == NULL)
{
   printf("**Error: couldn't allocate interface detail\n");
   SetupDiDestroyDeviceInfoList(hDeviceInfo);
   exit(-1);
}

pDeviceDetail->cbSize = sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);

// Get the detail info
if(!SetupDiGetDeviceInterfaceDetail(hDeviceInfo,
                                    &interfaceData,
                                    pDeviceDetail,
                                    requiredSize,
                                    NULL,
                                    NULL))
{
   printf("**Error: couldn't get interface detail(2), (%d)\n",
             GetLastError());

   SetupDiDestroyDeviceInfoList(hDeviceInfo);
   free(pDeviceDetail);
   exit(-1);
}

// Save the name
lstrcpyn(deviceName, pDeviceDetail->DevicePath, MAX_DEVICE_NAME);

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

```
// Open driver - Create the handle to the device
hRl1Base = CreateFile(deviceName,
                      GENERIC_READ    | GENERIC_WRITE,
                      FILE_SHARE_READ | FILE_SHARE_WRITE,
                      NULL,
                      OPEN_EXISTING,
                      NULL,
                      NULL);

if(hRl1Base == INVALID_HANDLE_VALUE)
{
   printf("**Error: couldn't open %s, (%d)\n", deviceName, GetLastError());
   exit(-1);
}

hDeviceInfo = SetupDiGetClassDevs(
                 (LPGUID)&GUID_DEVINTERFACE_RL1_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);

chan = 0;
i    = 0;

while(!done)
{// Find the interface for channel devices
   if(!SetupDiEnumDeviceInterfaces(hDeviceInfo,
                                   NULL,
                           (LPGUID)&GUID_DEVINTERFACE_RL1_CHAN,
                                   i,
                                   &interfaceData))
   {
      status = GetLastError();
      if(status == ERROR_NO_MORE_ITEMS)
      {
         printf("**Error: couldn't find device(no more items), (%d)\n",
                i);

         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
   hRl1Chan[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(hRl1Chan[chan] == INVALID_HANDLE_VALUE)
   {
      printf("**Error: couldn't open %s, (%d)\n",
              deviceName,
              GetLastError());

      SetupDiDestroyDeviceInfoList(hDeviceInfo);
      exit(-1);
   }

// Read info
   if( !DeviceIoControl(hRl1Chan[chan],
                        IOCTL_RL1_CHAN_GET_INFO,
                        NULL,
                        0,
                        &info,
                        sizeof(info),
                        &length,
                        NULL) )
   {
      printf("IOCTL_RL1_CHAN_GET_INFO failed:  %d\n", GetLastError());
      return -1;
   }

   if(info.InstanceNumber == devNum * RL1_BASE_NUM_CHANNELS + chan)
   {
      chan++;
      if(RL1_BASE_NUM_CHANNELS == chan)
         done = TRUE;
   }

   i++;
}

// Cleanup
SetupDiDestroyDeviceInfoList(hDeviceInfo);
```

IO Controls

The driver uses IO Control calls (IOCTLs) to configure the device. IOCTLs refer to a single Device Object which controls a single board or channel. IOCTLs are called using the Win32 function DeviceIoControl(), and passing in the handle to the device opened with CreateFile(). IOCTLs generally have input parameters, output parameters, or both. Often a custom structure is used. The IOCTLs defined in the RL1 drivers are described below:

IOCTL_RL1_BASE_GET_INFO

Function: Returns the device driver version, Xilinx flash revision, PLL device ID, user switch value, and device instance number.
Input: None
Output: RL1_BASE_DRIVER_DEVICE_INFO structure
Notes: The switch value is the configuration of the 8-bit onboard dipswitch that has been selected by the user (see the board silk screen for bit position and polarity). Instance number is the zero-based device number. See DDRl1Base.h for the definition of RL1_BASE_DRIVER_DEVICE_INFO.

IOCTL_RL1_BASE_LOAD_PLL_DATA

Function: Writes to the internal registers of the PLL.
Input: RL1_BASE_PLL_DATA structure
Output: None
Notes: The RL1_BASE_PLL_DATA structure has only one field: Data – an array of 40 bytes containing the PLL register data to write. See DDRl1Base.h for the definition of RL1_BASE_PLL_DATA.

IOCTL_RL1_BASE_READ_PLL_DATA

Function: Returns the contents of the internal registers of the PLL.
Input: None
Output: RL1_BASE_PLL_DATA structure
Notes: The register data is written to the RL1_BASE_PLL_DATA structure in an array of 40 bytes.

IOCTL_RL1_CHAN_GET_INFO

Function: Returns the channel driver version and the channel instance number.
Input: None
Output: RL1_CHAN_DRIVER_DEVICE_INFO structure
Notes: See DDRl1Base.h for the definition of RL1_CHAN_DRIVER_DEVICE_INFO.

IOCTL_RL1_CHAN_SET_CONFIG

Function: Specifies fields in the channel configuration register.
Input: RL1_CHAN_CONFIG structure
Output: None
Notes: This call controls channel configuration items that are not transmit or receive specific. The AutoDirSwitch field enables the automatic switching from transmit to receive and vice versa when the current active direction signals that it is done. When IoClockASel is true, PLL clock A is selected as the clock source, when it is false PLL clock B is selected. When ClockDiv is equal to one, the undivided clock source will be used for the 16x reference clock. Otherwise the clock source can be divided by any even number from two to thirty-two. See DDRl1Chan.h for the definition of RL1_CHAN_CONFIG.

IOCTL_RL1_CHAN_GET_STATE

Function: Returns the fields set in the previous call as well as the states of the master and read and write interrupt enables.
Input: None
Output: RL1_CHAN_STATE structure
Notes: The states of the interrupt enables are returned for informational purposes only. The values of these fields are controlled by other driver calls. The MIntEn field is the master interrupt enable for all user interrupts controlled by the EnableInterrupt and DisableInterrupt calls, whereas the WrDmaEn and RdDmaEn fields are automatically controlled by the driver in response to WriteFile and ReadFile calls. See DDRl1Chan.h for the definition of RL1_CHAN_STATE.

IOCTL_RL1_CHAN_GET_STATUS

Function: Returns the channel's status register value and clears the latched status bits.
Input: None
Output: Value of the channel's status register (unsigned long integer)
Notes: See DDRl1Chan.h for the status bit definitions. Only the bits in STATUS_MASK will be returned. The bits in STATUS_LATCH_MASK will be cleared by this call only if they are set when the register was read. This prevents the possibility of missing an interrupt condition that occurs after the register has been read but before the latched register bits are cleared.

IOCTL_RL1_CHAN_SET_FIFO_LEVELS

Function: Sets the channel's receiver almost full and transmitter almost empty levels.
Input: RL1_CHAN_FIFO_LEVELS structure
Output: None
Notes: These FIFO levels are used to determine TX almost empty and RX almost full status when the FIFO data counts reach the specified levels. They are also used to signal priority for the DMA request/grant arbiter, if this has been enabled for the referenced channel. See DDRl1Chan.h for the definition of RL1_CHAN_FIFO_LEVELS.

IOCTL_RL1_CHAN_GET_FIFO_LEVELS

Function: Returns the channel's receiver almost full and transmitter almost empty levels.
Input: None
Output: RL1_CHAN_FIFO_LEVELS structure
Notes: See DDRl1Chan.h for the definition of RL1_CHAN_FIFO_LEVELS.

IOCTL_RL1_CHAN_WRITE_FIFO

Function: Writes a single 32-bit word to the channel's transmit FIFO.
Input: FIFO data word (unsigned long integer)
Output: None
Notes: Normally the write command is used to load data into the device. This call can be used for small amounts of data, but is inefficient for larger sized transfers.

IOCTL_RL1_CHAN_READ_FIFO

Function: Reads a single 32-bit word from the channel's receive FIFO.
Input: None
Output: FIFO data word (unsigned long integer)
Notes: Normally the read command is used to retrieve data from the device. This call can be used for small amounts of data, but is inefficient for larger sized transfers.

IOCTL_RL1_CHAN_GET_FIFO_COUNTS

Function: Returns the number of data words in the transmit and receive FIFOs.
Input: None
Output: RL1_CHAN_FIFO_COUNTS structure
Notes: Returns the number of words in the referenced channels I/O data circuitry. For the transmitter this is a maximum of one more than the FIFO size and for the receiver the data-count can be as much as four words more than the FIFO size. The excess is due to data pipe-line latches in the I/O data-path. See DDRl1Chan.h for the definition of RL1_CHAN_FIFO_COUNTS.

IOCTL_RL1_CHAN_RESET_FIFOS

Function: Resets the Tx and/or Rx FIFOs for the referenced channel.
Input: RL1_FIFO_SEL enumerated type
Output: None
Notes: See DDRI1Chan.h for the definition of RL1_FIFO_SEL.

IOCTL_RL1_CHAN_SET_TX_CONFIG

Function: Specifies various parameters that control the behavior of the transmitter.
Input: RL1_CHAN_TX_CONFIG structure
Output: None
Notes: See DDRI1Chan.h for the definition of RL1_CHAN_TX_CONFIG.

IOCTL_RL1_CHAN_GET_TX_STATE

Function: Returns the parameters set in the previous call as well as the state of the transmitter enable bit.
Input: None
Output: RL1_CHAN_TX_STATE structure
Notes: If the ClearEnable field has been set to true, the Enabled field can be monitored to indicate when the current message has completed. See DDRI1Chan.h for the definition of RL1_CHAN_TX_STATE.

IOCTL_RL1_CHAN_SET_RX_CONFIG

Function: Specifies various parameters that control the behavior of the receiver.
Input: RL1_CHAN_RX_CONFIG structure
Output: None
Notes: TermEnable activates the 100Ω shunt termination on the receive data lines. When the interface is operating in half-duplex mode, the termination will only be active when the transmitter is not active. See DDRI1Chan.h for the definition of RL1_CHAN_RX_CONFIG.

IOCTL_RL1_CHAN_GET_RX_STATE

Function: Returns the parameters set in the previous call as well as the state of the receiver enable bit.
Input: None
Output: RL1_CHAN_RX_STATE structure
Notes: If the ClearEnable field has been set to true, the Enabled field can be monitored to indicate when the current message has completed. See DDRI1Chan.h for the definition of RL1_CHAN_RX_STATE.

IOCTL_RL1_CHAN_START_TX

Function: Starts a data transmission provided valid data is available to send.
Input: Number of bytes to send (unsigned short integer)
Output: None
Notes: If the input field is NULL or zero, the transmission will continue until all FIFO data has been sent. If this field is non-zero, only the specified number of bytes will be sent.

IOCTL_RL1_CHAN_STOP_TX

Function: Abort or cancel a data transmission.
Input: None
Output: None
Notes: This call will cancel a transmit request that has not started or stop a transmission in progress.

IOCTL_RL1_CHAN_START_RX

Function: Enable the receiver to look for data and store it in the receive FIFO.
Input: None
Output: None
Notes: .

IOCTL_RL1_CHAN_STOP_RX

Function: Abort or cancel a data reception.
Input: None
Output: None
Notes: This call will cancel a receive request that has not started or stop a reception in progress.

IOCTL_RL1_CHAN_GET_RX_BYTE_COUNT

Function: Returns the number of bytes received in the last message.
Input: None
Output: Received byte count (unsigned short integer)
Notes: Each channel contains a 16-bit counter that increments each time a data byte is received. When the received data input is high for at least 8 bit-periods after the end of a data-byte, the receiver sets the STAT_RX_INT status bit, transfers this count to the byte-count register and clears the counter for the next message. The byte-count register value is returned by this call. The value will remain valid until the end of a subsequent message.

IOCTL_RL1_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_RL1_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 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 the interrupts.

IOCTL_RL1_CHAN_DISABLE_INTERRUPT

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

IOCTL_RL1_CHAN_FORCE_INTERRUPT

Function: Causes a system interrupt to occur.
Input: None
Output: None
Notes: Causes an interrupt to be asserted on the PCI bus if the master interrupt is enabled. This IOCTL is used for test and development, to test interrupt processing.

IOCTL_RL1_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 status that was read in the interrupt service routine for the last user interrupt serviced. Latched status bits (bits in STATUS_LATCH_MASK) that were set when the status was read in the ISR are returned along with the other status bits, but will have been automatically cleared in the interrupt DPC.

Write

PMC-BiSerial-III RL1 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 target device, a pointer to a pre-allocated buffer containing the data to be written, an unsigned long integer that represents the number of bytes to be transferred, a pointer to an unsigned long integer to contain the number of bytes actually written, and a pointer to an optional Overlapped structure for performing asynchronous I/O.

Read

PMC-BiSerial-III RL1 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 target device, a pointer to a pre-allocated buffer that will contain the data read, an unsigned long integer that represents the number of bytes to be transferred, a pointer to an unsigned long integer to contain the number of bytes actually read, and a pointer to an optional Overlapped structure for performing asynchronous I/O.

Warranty and Repair

Dynamic Engineering warrants this product to be free from defects under normal use and service and in its original, unmodified condition, for a period of one year from the time of purchase. If the product is found to be defective within the terms of this warranty, Dynamic Engineering's sole responsibility shall be to repair, or at Dynamic Engineering's sole option to replace, the defective product.

Dynamic Engineering's warranty of and liability for defective products is limited to that set forth herein. Dynamic Engineering disclaims and excludes all other product warranties and product liability, expressed or implied, including but not limited to any implied warranties of merchandisability or fitness for a particular purpose or use, liability for negligence in manufacture or shipment of product, liability for injury to persons or property, or for any incidental or consequential damages.

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

Service Policy

Before returning a product for repair, verify as well as possible that the driver is at fault. The driver has gone through extensive testing and in most cases it will be "cockpit error" rather than an error with the driver. When you are sure or at least willing to pay to have someone help then call the Customer Service Department and arrange to speak with an engineer. We will work with you to determine the cause of the issue. If the issue is one of a defective driver we will correct the problem and provide an updated module(s) to you [no cost]. If the issue is of the customer's making [anything that is not the driver] the engineering time will be invoiced to the customer. Pre-approval may be required in some cases depending on the customer's invoicing policy.

Out of Warranty Repairs

Out of warranty support will be billed. The current minimum repair charge is $125. An open PO will be required.

For Service Contact:

Customer Service Department
Dynamic Engineering
150 DuBois, Suite C Santa Cruz, CA 95060
(831) 457-8891 Fax (831) 457-4793
support@dyneng.com