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

HLnkBase
&
HLnkChan

Driver Documentation

Win32 Driver Model

Revision A
Corresponding Hardware: Revision A
10-2009-0101

HLnkBase & HLnkChan
WDM Device Drivers for the ccPMC-HOTLink 6-Channel HOTLink Conduction-cooled PMC module

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

Manual Revision A Revised April 17, 2009.

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

The HLnkBase and HLnkChan drivers are Win32 driver model (WDM) device drivers for the PMC-HOTLink from Dynamic Engineering.

The HOTLink board has a Spartan3-4000 Xilinx FPGA to implement the PCI interface, FIFOs and protocol control and status for six HOTLink channels. There is also a programmable PLL with two clock outputs to create separate programmable I/O clocks for the HOTLink I/O and the RS-485 I/O. Each channel has an 4k by 32-bit receive and a 2k by 32-bit transmit data FIFO for the HOTLink I/O and a 1k by 32-bit data FIFO, for each of the two bidirectional RS-485 lines.

The HLnkChan driver WriteFile() call will initiate a DMA transfer into the HOTLink transmit FIFO and the ReadFile() call will initiate a DMA transfer from the HOTLink receive FIFO. The RS-485 FIFOs are only accessed by single 32-bit word transfers.

When the HOTLink board is recognized by the PCI bus configuration utility it will start the HLnkBase driver which will create a device object for each board, initialize the hardware; create child devices for the six I/O channels and request loading of the HLnkChan driver. The HLnkChan driver will create a device object for each of the I/O channels and perform initialization on each channel. 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.

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 HOTLink user manual (also referred to as the hardware manual).

Driver Installation

There are several files provided in each driver package. These files include HOTLink.inf, HLnkBase.sys, DDHLnkBase.h, HLnkBaseGUID.h, HLnkChan.sys, DDHLnkChan.h, HLnkChanGUID.h, HLnkTest.exe, and HLnkTest source files.

HLnkBaseGUID.h and HLnkChanGUID.h are C header files that define the device interface identifiers for the drivers. DDHLnkBase.h and DDHLnkChan.h files are C header files that define the Application Program Interface (API) to 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.

HLnkTest.exe is a sample Win32 console applications that makes calls into the HLnkBase/HLnkChan drivers to test each driver call without actually writing any application code. They are not required during driver installation either.

To run HLnkTest, open a command prompt console window and type ***HLnkTest -d0 -?*** to display a list of commands (the HLnkTest.exe file must be in the directory that the window is referencing). The commands are all of the form ***HLnkTest -dn -im*** where ***n*** and ***m*** are the device number and HLnkBase driver ioctl number respectively or ***HLnkTest -cn -im*** where ***n*** and ***m*** are the channel number (0-5) and HLnkChan driver ioctl number respectively.

This test application is intended to test the proper functioning of each driver call, **not** for normal operation.

Windows 2000 Installation

Copy HOTLink.inf, HLnkBase.sys and HLnkChan.sys to a floppy disk, CD, or other accessible location.

With the HOTLink 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 HOTLink.inf file.
_ Select ***Next***
_ Select ***Finish*** to close the ***Found New Hardware Wizard***.
The system should now see the HOTLink channels and reopen the ***New Hardware Wizard.*** Proceed as above for each channel as necessary.

Windows XP Installation

Copy HOTLink.inf, HLnkBase.sys and HLnkChan.sys to a floppy disk, CD, or other accessible location.

With the HOTLink 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 HOTLink channels and reopen the ***New Hardware Wizard.*** Proceed as above for each channel as necessary.

Driver Startup

Once the drivers have been installed they 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 names obtained from the system.

The interfaces to the devices are identified using globally unique identifiers (GUIDs), which are defined in HLnkBaseGUID.h and HLnkChanGUID.h.

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

```
// The maximum length of the device name for a given interface
#define MAX_DEVICE_NAME 256
// Handles to the device objects
HANDLE hHLnkBase                         =  INVALID_HANDLE_VALUE;

HANDLE hHLnkChan[HLNK_BASE_NUM_CHANNELS] = {INVALID_HANDLE_VALUE,
                                            INVALID_HANDLE_VALUE,
                                            INVALID_HANDLE_VALUE,
                                            INVALID_HANDLE_VALUE,
                                            INVALID_HANDLE_VALUE,
                                            INVALID_HANDLE_VALUE};
// HOTLink device number
ULONG                            devNum
// HOTLink channel handle array index and interface number
ULONG                            chan, i;
// 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 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;
// The base device information structure
HLNK_BASE_DRIVER_DEVICE_INFO     info;
// The channel device information structure
HLNK_CHAN_DRIVER_DEVICE_INFO     cinfo;
// Flag indicating success finding correct device
BOOLEAN                          found = FALSE;

hDeviceInfo = SetupDiGetClassDevs(
                       (LPGUID)&GUID_DEVINTERFACE_HLNK_BASE,
                               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);

i = 0;
while(!found)
{// Find the interface for device devNum
   if(!SetupDiEnumDeviceInterfaces(hDeviceInfo,
                                   NULL,
                           (LPGUID)&GUID_DEVINTERFACE_HLNK_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);
      }
   }

 // 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
  hHLnkBase = CreateFile(deviceName,
                         GENERIC_READ    | GENERIC_WRITE,
                         FILE_SHARE_READ | FILE_SHARE_WRITE,
                         NULL,
                         OPEN_EXISTING,
                         NULL,
                         NULL);

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

     exit(-1);
  }
```

```
 // Read info
   if(!DeviceIoControl(hHLnkBase,
                       IOCTL_HLNK_BASE_GET_INFO,
                       NULL,
                       0,
                       &info,
                       sizeof(info),
                       &length,
                       NULL))
   {
      printf("IOCTL_HLNK_BASE_GET_INFO failed:  %d\n", GetLastError());
      exit(-1);
   }

   if(info.InstanceNumber == devNum)
      found = TRUE;
   else
      i++;
}

SetupDiDestroyDeviceInfoList(hDeviceInfo);

hDeviceInfo = SetupDiGetClassDevs(
                       (LPGUID)&GUID_DEVINTERFACE_HLNK_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);

i    = 0;
chan = 0;

while(chan < HLNK_BASE_NUM_CHANNELS)
{// Find the interface for device
   if(!SetupDiEnumDeviceInterfaces(hDeviceInfo,
                                   NULL,
                          (LPGUID)&GUID_DEVINTERFACE_HLNK_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
   hHLnkChan[chan] = CreateFile(deviceName,
                                GENERIC_READ    | GENERIC_WRITE,
                                FILE_SHARE_READ | FILE_SHARE_WRITE,
                                NULL,
                                OPEN_EXISTING,
                                NULL,
                                NULL);

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

   if(!DeviceIoControl(hHLnkChan[chan],
                       IOCTL_HLNK_CHAN_GET_INFO,
                       NULL,
                       0,
                       &cinfo,
                       sizeof(cinfo),
                       &length,
                       NULL) )
   {
      printf("IOCTL_HLNK_CHAN_GET_INFO failed:  %d\n", GetLastError());
      exit(-1);
   }

   if(cinfo.InstanceNumber / HLNK_BASE_NUM_CHANNELS == devNum &&
      cinfo.InstanceNumber % HLNK_BASE_NUM_CHANNELS == chan)
   {
      chan++;
   }

   i++;
}
```

IO Controls

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

```
BOOL DeviceIoControl(
  HANDLE       hDevice,          // Handle opened with CreateFile()
  DWORD        dwIoControlCode,  // Control code defined in API header file
  LPVOID       lpInBuffer,       // Pointer to input parameter
  DWORD        nInBufferSize,    // Size of input parameter
  LPVOID       lpOutBuffer,      // Pointer to output parameter
  DWORD        nOutBufferSize,   // Size of output parameter
  LPDWORD      lpBytesReturned,  // Pointer to return length parameter
  LPOVERLAPPED lpOverlapped,     // Optional pointer to overlapped structure
);                               //   used for asynchronous I/O
```

The IOCTLs defined for the HLnkBase driver are described below:

IOCTL_HLNK_BASE_GET_INFO

Function: Returns the Driver version, Xilinx revision, Switch value, Instance number, and PLL ID.
Input: None
Output: HLNK_BASE_DRIVER_DEVICE_INFO structure
Notes: Switch value is the configuration of the on-board dip-switch that has been set by the User (see the board silk screen for bit position and polarity). The PLL ID is the device address of the PLL device. This value, which is set at the factory, is usually 0x69 but may also be 0x6A. See DDHLnkBase.h for the definition of HLNK_BASE_DRIVER_DEVICE_INFO.

IOCTL_HLNK_BASE_LOAD_PLL_DATA

Function: Loads the internal registers of the PLL.
Input: HLNK_BASE_PLL_DATA structure
Output: None
Notes: The PLL internal register data is loaded into the HLNK_BASE_PLL_DATA structure in an array of 40 bytes. Appropriate values for this array can be derived from .jed files created by the CyberClock utility from Cypress Semiconductor.

IOCTL_HLNK_BASE_READ_PLL_DATA

Function: Returns the contents of the PLL's internal registers
Input: None
Output: HLNK_BASE_PLL_DATA structure
Notes: Data is in an array of 40 bytes in the HLNK_BASE_PLL_DATA structure.

The IOCTLs defined for the HLnkChan driver are described below:

IOCTL_HLNK_CHAN_GET_INFO

Function: Returns the driver version and instance number of the referenced channel.
Input: None
Output: HLNK_CHAN_DRIVER_DEVICE_INFO structure
Notes: See the definition of HLNK_CHAN_DRIVER_DEVICE_INFO in the DDHLnkChan.h header file.

IOCTL_HLNK_CHAN_SET_CONFIG
Function: Writes a configuration value to the channel control register.
Input: Value of channel control register (unsigned long integer)
Output: None
Notes: See DDHLnkChan.h for the relevant channel control bit definitions. Only the bits in CHAN_CNTRL_MASK can be controlled by this call.

IOCTL_HLNK_CHAN_GET_CONFIG
Function: Returns the channel's control configuration.
Input: None
Output: Value of the channel control register (unsigned long integer)
Notes: Returns the values of the bits in CHAN_CNTRL_READ_MASK.

IOCTL_HLNK_CHAN_GET_STATUS
Function: Returns the channel's status value and clears the latched bits.
Input: None
Output: Value of channel status register (unsigned long integer)
Notes: The latched bits in CHAN_STAT_LATCH_MASK will be cleared if they are set when the status is read.

IOCTL_HLNK_CHAN_SET_FIFO_LEVELS
Function: Sets the transmitter almost empty and receiver almost full levels for the channel.
Input: HLNK_CHAN_FIFO_LEVELS structure
Output: None
Notes: These values are initialized to the default values _ FIFO and _ FIFO respectively when the driver initializes. The FIFO counts are compared to these levels to determine the value of the CHAN_STAT_TX_FF_AMT and CHAN_STAT_RX_FF_AFL status bits. Also, if the control bits CHAN_CNTRL_URGNT_IN_EN and/or CHAN_CNTRL_URGNT_OUT_EN are set, these levels are used to determine when to give priority to an input or output DMA channel.

IOCTL_HLNK_CHAN_GET_FIFO_LEVELS
Function: Returns the transmitter almost empty and receiver almost full levels for the channel.
Input: None
Output: HLNK_CHAN_FIFO_LEVELS structure
Notes:

IOCTL_HLNK_CHAN_GET_FIFO_COUNTS
Function: Returns the number of data words in the transmit and receive FIFOs.
Input: None
Output: HLNK_CHAN_FIFO_COUNTS structure
Notes: There is one pipe-line latch for the transmit FIFO data and four for the receive FIFO data. These are counted in the FIFO counts. That means the transmit count can be a maximum of 2049 32-bit words and the receive count can be a maximum of 4100 32-bit words.

IOCTL_HLNK_CHAN_RESET_FIFOS
Function: Resets one or both FIFOs for the referenced channel.
Input: HLNK_FIFO_SEL enumeration type
Output: None
Notes: Resets the transmit or receive FIFO or both depending on the input parameter selection. Also sets the programmable almost full/empty levels back to the default values for the FIFO(s) that were reset.

IOCTL_HLNK_CHAN_WRITE_FIFO
Function: Writes a 32-bit data-word to the transmit FIFO.
Input: FIFO word (unsigned long integer)
Output: None
Notes: Used to make single-word accesses to the transmit FIFO instead of using DMA.

IOCTL_HLNK_CHAN_READ_FIFO
Function: Returns a 32-bit data word from the receive FIFO.
Input: None
Output: FIFO word (unsigned long integer)
Notes: Used to make single-word accesses to the receive FIFO instead of using DMA.

IOCTL_HLNK_CHAN_SET_485_CONFIG
Function: Writes a configuration value to the channel RS-485 control register.
Input: Value of channel RS-485 control register (unsigned long integer)
Output: None
Notes: See DDHLnkChan.h for the relevant channel RS-485 control bit definitions. Only the bits in CHAN_485_CNTRL_MASK can be controlled by this call.

IOCTL_HLNK_CHAN_GET_485_CONFIG
Function: Returns the channel's RS-485 control configuration.
Input: None
Output: Value of the channel RS-485 control register (unsigned long integer)
Notes: Returns the values of the bits in CHAN_485_CNTRL_MASK.

IOCTL_HLNK_CHAN_GET_485_STATUS
Function: Returns the channel's RS-485 status register value.
Input: None
Output: Value of channel RS-485 status register (unsigned long integer)
Notes:

IOCTL_HLNK_CHAN_RESET_485_FIFOS
Function: Resets one or both RS-485 FIFOs for the channel.
Input: HLNK_CHAN_485_FIFO_SEL enumeration type
Output: None
Notes: Resets the RS-485A or RS-485B FIFO or both depending on the input parameter selection.

IOCTL_HLNK_CHAN_WRITE_485A_FIFO
Function: Writes a 32-bit data-word to the RS-485A FIFO.
Input: FIFO word (unsigned long integer)
Output: None
Notes: Used to write data to the RS-485A FIFO.

IOCTL_HLNK_CHAN_READ_485A_FIFO
Function: Returns a 32-bit data word from the RS-485A FIFO.
Input: None
Output: FIFO word (unsigned long integer)
Notes: Used to read data from the RS-485A FIFO.

IOCTL_HLNK_CHAN_WRITE_485B_FIFO
Function: Writes a 32-bit data-word to the RS-485B FIFO.
Input: FIFO word (unsigned long integer)
Output: None
Notes: Used to write data to the RS-485B FIFO.

IOCTL_HLNK_CHAN_READ_485B_FIFO
Function: Returns a 32-bit data word from the RS-485B FIFO.
Input: None
Output: FIFO word (unsigned long integer)
Notes: Used to read data from the RS-485B FIFO.

IOCTL_HLNK_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_HLNK_CHAN_ENABLE_INTERRUPT

Function: Enables the channel 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 when a user interrupt is serviced. Therefore this command must be run after each interrupt occurs to re-enable it.

IOCTL_HLNK_CHAN_DISABLE_INTERRUPT

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

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

IOCTL_HLNK_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 of the last interrupt caused by one of the enabled channel interrupts. The interrupts that deal with the DMA transfers do not affect this value.

Write

HOTLink DMA data is written to the referenced I/O channel device using the write command. Writes are executed using the Win32 function WriteFile() and passing in the handle to the I/O channel 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

HOTLink DMA data is read from the referenced I/O channel device using the read command. Reads are executed using the Win32 function ReadFile() and passing in the handle to the I/O channel 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 Street, Suite C
Santa Cruz, CA 95060
831-457-8891
831-457-4793 Fax
support@dyneng.com