

## **DYNAMIC ENGINEERING**

150 DuBois, Suite 3 Santa Cruz, CA 95060

(831) 457-8891 **Fax** (831) 457-4793

<http://www.dyneng.com>

[sales@dyneng.com](mailto:sales@dyneng.com)

Est. 1988

# **HpnBase & HpnChan**

## **Driver Documentation**

### **Win32 Driver Model**

Revision B

Corresponding Hardware: Revision A

10-2007-0601

Corresponding Firmware: Revision C

**HpnBase & HpnChan**  
WDM Device Drivers for the  
PCI-Harpoon 4-Channel Simulator and  
Digital Data-Link Serial Interface

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

©2008 by Dynamic Engineering.  
Other trademarks and registered trademarks are  
owned by their respective manufactures.  
Manual Revision B. Revised March 14, 2008

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.....	12
IOCTL_HP_N_BASE_GET_INFO.....	12
IOCTL_HP_N_BASE_LOAD_PLL_DATA.....	12
IOCTL_HP_N_BASE_READ_PLL_DATA.....	13
IOCTL_HP_N_CHAN_GET_INFO .....	14
IOCTL_HP_N_CHAN_SET_CONFIG .....	14
IOCTL_HP_N_CHAN_GET_STATE .....	14
IOCTL_HP_N_CHAN_GET_STATUS.....	14
IOCTL_HP_N_CHAN_CLR_STATUS.....	14
IOCTL_HP_N_CHAN_PUT_TX_WORD .....	15
IOCTL_HP_N_CHAN_GET_RX_WORD .....	15
IOCTL_HP_N_CHAN_SEND_WORD.....	15
IOCTL_HP_N_CHAN_SET_TEST_SIGS .....	15
IOCTL_HP_N_CHAN_SET_VOLTAGE_CONFIG.....	15
IOCTL_HP_N_CHAN_GET_VOLTAGE_CONFIG .....	15
IOCTL_HP_N_CHAN_REGISTER_EVENT.....	16
IOCTL_HP_N_CHAN_ENABLE_INTERRUPT.....	16
IOCTL_HP_N_CHAN_DISABLE_INTERRUPT.....	16
IOCTL_HP_N_CHAN_FORCE_INTERRUPT.....	16
IOCTL_HP_N_CHAN_GET_ISR_STATUS .....	16
Warranty and Repair.....	17
Service Policy.....	17
Out of Warranty Repairs .....	17
For Service Contact:.....	17

## Introduction

The HpnBase and HpnChan drivers are Win32 driver model (WDM) device drivers for the PCI-Harpoon from Dynamic Engineering. The PCI-Harpoon has a Spartan3-1500 Xilinx FPGA to implement the PCI interface, protocol control and status for four Harpoon missile simulators and bidirectional digital data-link channels. The board also has an onboard programmable PLL to provide the 800 kHz timing reference and twenty-four RS-485 line driver/receivers for the external serial interface (data, clock and enable in and out).

When the PCI-Harpoon is recognized by the PCI bus configuration utility it will start the HpnBase driver. The HpnBase driver enumerates the four I/O channels and creates separate HpnChan 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, input and output data-words and read status.

## 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 PCI-Harpoon user manual (also referred to as the hardware manual).

## Driver Installation

There are several files provided the driver package. These files include HpnBase.sys, HpnChan.sys, PciHarpoon.inf, DDHpnBase.h, DDHpnChan.h, HpnBaseGUID.h, HpnChanGUID.h, HpnTest.exe, and HpnTest source files.

DDHpnBase.h and DDHpnChan.h are C header files that defines the Application Program Interface (API) to the drivers. HpnBaseGUID.h, HpnChanGUID.h 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. HpnBase.sys and HpnChan.sys are the actual driver executables that are loaded into kernel memory to provide the software/hardware interface to control the PCI-Harpoon. PciHarpoon.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 PCI-Harpoon.

HpnTest.exe is a sample Win32 console application that makes calls into the HpnBase and HpnChan drivers to test each driver call without actually writing any application code. It is not required for driver installation.



To run HpnTest.exe, open a command prompt console window and type **HpnTest -d0 - ?** to display a list of commands (the HpnTest.exe file must be in the directory that the window is referencing). The commands are all of the form **HpnTest -dn -im** where **n** and **m** are the zero-based device instance number and driver ioctl number respectively or **HpnTest -cn -im** where **n** and **m** are the zero-based 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 PciHarpoon.inf, HpnBase.sys and HpnChan.sys to a floppy disk, or CD if preferred.

With the PCI-Harpoon 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 PciHarpoon.inf file.
- Select **Next**.
- Select **Finish** to close the **Found New Hardware Wizard**.

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

## Windows XP Installation

Copy PciHarpoon.inf, HpnBase.sys and HpnChan.sys to a floppy disk, or CD if preferred.

With the PCI-Harpoon 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 PCI-Harpoon 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 HpnBaseGUID.h and HpnChanGUID.h. Below is example code for opening handles for device devNum.

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

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

// Handles to device objects
HANDLE hHpnBase = INVALID_HANDLE_VALUE;
HANDLE hHpnChan[HPN_BASE_NUM_CHANNELS] = {INVALID_HANDLE_VALUE,
                                           INVALID_HANDLE_VALUE,
                                           INVALID_HANDLE_VALUE,
                                           INVALID_HANDLE_VALUE};

// PCI-Harpoon device number (starting with zero)
ULONG devNum;

// Hpn 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
HPN_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 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_HP_N_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_HP_N_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
hHpnBase = CreateFile(deviceName,
                      GENERIC_READ   | GENERIC_WRITE,
                      FILE_SHARE_READ | FILE_SHARE_WRITE,
                      NULL,
                      OPEN_EXISTING,
                      NULL,
                      NULL);

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

```

```

hDeviceInfo = SetupDiGetClassDevs(
    (LPGUID)&GUID_DEVINTERFACE_HP_N_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_HP_N_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
hHpnChan[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(hHpnChan[chan] == INVALID_HANDLE_VALUE)
{
    printf("***Error: couldn't open %s, (%d)\n",
           deviceName,
           GetLastError());

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

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

if(info.InstanceNumber == devNum * HPN_BASE_NUM_CHANNELS + chan)
{
    chan++;
    if(HPN_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. 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
```

### IOCTL\_HP\_N\_BASE\_GET\_INFO

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

**Input:** None

**Output:** HP\_N\_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 DDHpnBase.h for the definition of HP\_N\_BASE\_DRIVER\_DEVICE\_INFO.

### IOCTL\_HP\_N\_BASE\_LOAD\_PLL\_DATA

**Function:** Loads the internal registers of the PLL.

**Input:** HP\_N\_BASE\_PLL\_DATA structure

**Output:** None

**Notes:** The HP\_N\_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\_HP\_N\_BASE\_READ\_PLL\_DATA**

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

**Input:** None

**Output:** HPN\_BASE\_PLL\_DATA structure

**Notes:** The register data is output in the HPN\_BASE\_PLL\_DATA structure in an array of 40 bytes.

## **IOCTL\_HP\_N\_CHAN\_GET\_INFO**

**Function:** Returns the Driver version and Instance number.

**Input:** None

**Output:** HPN\_CHAN\_DRIVER\_DEVICE\_INFO structure

**Notes:** Instance number is the zero-based device number. See DDHpnChan.h for the definition of HPN\_CHAN\_DRIVER\_DEVICE\_INFO.

## **IOCTL\_HP\_N\_CHAN\_SET\_CONFIG**

**Function:** Configures the channel control register for an I/O channel on the PCI-Harpoon.

**Input:** HPN\_CHAN\_CONFIG structure

**Output:** None

**Notes:** Sets the configuration parameters for the channel interface, including state-machine enables, interrupt mode, termination enables and the receive data invert (used for testing data loop-back). See DDHpnChan.h for the definition of HPN\_CHAN\_CONFIG. Also see the hardware manual for detailed descriptions of the control parameter functions.

## **IOCTL\_HP\_N\_CHAN\_GET\_STATE**

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

**Input:** None

**Output:** HPN\_CHAN\_STATE structure

**Notes:** Returns the fields set in the previous call as well as the state of the master interrupt enable. This command is used mainly for testing or for saving the channel state for later restoration.

## **IOCTL\_HP\_N\_CHAN\_GET\_STATUS**

**Function:** Returns the value of the channel status register.

**Input:** None

**Output:** Value of status register (unsigned long integer)

**Notes:** See DDHpnChan.h for the status bit definitions. The interrupt status bits will be latched even if they are not enabled, but only the enabled interrupts will cause a system interrupt to occur.

## **IOCTL\_HP\_N\_CHAN\_CLR\_STATUS**

**Function:** Clears the latched status bits.

**Input:** Value of status bits to clear (unsigned long integer)

**Output:** None

**Notes:** See DDHpnChan.h for the status bit definitions. The bits in STAT\_LATCH\_MASK will be cleared if they are set high in the input field to this call.

### **IOCTL\_HP\_N\_CHAN\_PUT\_TX\_WORD**

**Function:** Writes the 17-bit data-word (16-bit data plus parity) to the shift register.

**Input:** Transmit data value (unsigned long integer)

**Output:** None

**Notes:**

### **IOCTL\_HP\_N\_CHAN\_GET\_RX\_WORD**

**Function:** Reads the 17-bit data word from the shift register.

**Input:** None

**Output:** Receive data value (unsigned long integer)

**Notes:**

### **IOCTL\_HP\_N\_CHAN\_SEND\_WORD**

**Function:** Causes the Test Enable and Test Clock outputs to send 17-data clocks framed by an active-high enable signal.

**Input:** None

**Output:** None

**Notes:**

### **IOCTL\_HP\_N\_CHAN\_SET\_TEST\_SIGS**

**Function:** Explicitly controls the Test Enable and Test Clock outputs.

**Input:** HPN\_CHAN\_TEST\_SIGS structure

**Output:** None

**Notes:** The structure has two Boolean fields: Enable and Clock. When these are true, the respective output signal is set high; when false, they are set low. See DDHpnChan.h for the definition of HPN\_CHAN\_TEST\_SIGS.

### **IOCTL\_HP\_N\_CHAN\_SET\_VOLTAGE\_CONFIG**

**Function:** Controls the opto-isolated switches that control the 28-volt outputs and returns.

**Input:** HPN\_CHAN\_VOLT\_CONFIG structure

**Output:** None

**Notes:** There are seven +28-volt switches and four ground switches. See DDHpnChan.h for the definition of HPN\_CHAN\_VOLT\_CONFIG.

### **IOCTL\_HP\_N\_CHAN\_GET\_VOLTAGE\_CONFIG**

**Function:** Returns the bits set in the previous call.

**Input:** None

**Output:** HPN\_CHAN\_VOLT\_CONFIG structure

**Notes:**



## **IOCTL\_HP\_N\_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\_HP\_N\_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.

## **IOCTL\_HP\_N\_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\_HP\_N\_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\_HP\_N\_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 value of the status register that was read in the interrupt service routine of the last interrupt serviced. The STAT\_TX\_INT and STAT\_RX\_INT status bits will only be returned if they were enabled in the IOCTL\_HP\_N\_CHAN\_SET\_CONFIG call. See DDHpnChan.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.

## 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](mailto:support@dyneng.com)

All information provided is Copyright Dynamic Engineering.

