

DYNAMIC ENGINEERING

435 Park Dr., Ben Lomond, Calif. 95005

831-336-8891 Fax 831-336-3840

<http://www.dyneng.com>

sales@dyneng.com

Est. 1988

IpQuart

Driver Documentation

Win32 Driver Model

Revision A

Corresponding Hardware: Revision B/D

10-2002-0202/0204

Corresponding Firmware: Revision B/C

IpQuart
WDM Device Drivers for the
IP-QuadUART IP Module

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

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.

This product has been designed to operate with IP Module carriers and compatible user-provided equipment. Connection of incompatible hardware is likely to cause serious damage.

©2005 by Dynamic Engineering.

Trademarks and registered trademarks are owned by their respective manufactures.
Manual Revision A. Revised January 11, 2005.



Table of Contents

Introduction	5
Note	5
Driver Installation	5
Windows 2000 Installation	5
Windows XP Installation	6
Driver Startup	7
IO Controls	9
IOCTL_IPQUART_GET_INFO	9
IOCTL_IPQUART_SET_ACTIVE_CHAN	9
IOCTL_IPQUART_SET_IP_CONTROL	10
IOCTL_IPQUART_GET_IP_CONTROL	10
IOCTL_IPQUART_SET_BASE_CONFIG	10
IOCTL_IPQUART_GET_BASE_CONFIG	10
IOCTL_IPQUART_SET_CHAN_CONFIG	10
IOCTL_IPQUART_GET_CHAN_CONFIG	11
IOCTL_IPQUART_SET_UART_DATA_CONFIG	11
IOCTL_IPQUART_GET_UART_DATA_CONFIG	11
IOCTL_IPQUART_SET_UART_INTEN	11
IOCTL_IPQUART_GET_UART_INTEN	11
IOCTL_IPQUART_SET_UART_MODEM_CONTROL	12
IOCTL_IPQUART_GET_UART_MODEM_CONTROL	12
IOCTL_IPQUART_SET_UART_FLOW_CONTROL_PARAMS	12
IOCTL_IPQUART_SET_UART_FLOW_CONTROL_MODE	12
IOCTL_IPQUART_GET_UART_FLOW_CONTROL_MODE	13
IOCTL_IPQUART_WRITE_UART_DATA_BYTE	13
IOCTL_IPQUART_READ_UART_DATA_BYTE	13
IOCTL_IPQUART_SET_TIMEOUT_CONFIG	13
IOCTL_IPQUART_GET_TIMEOUT_CONFIG	13
IOCTL_IPQUART_RESET_UART	14
IOCTL_IPQUART_CONFIGURE_UART_FIFOS	14
IOCTL_IPQUART_GET_UART_STATUS	14
IOCTL_IPQUART_GET_STATUS	14
IOCTL_IPQUART_REGISTER_EVENT	15
IOCTL_IPQUART_ENABLE_INTERRUPT	15
IOCTL_IPQUART_DISABLE_INTERRUPT	15
IOCTL_IPQUART_FORCE_INTERRUPT	15
IOCTL_IPQUART_SET_VECTOR	16
IOCTL_IPQUART_GET_VECTOR	16



IOCTL_IPQUART_GET_ISR_STATUS	16
Write	17
Read	17
WARRANTY AND REPAIR	17
Service Policy	18
Out of Warranty Repairs	18
For Service Contact:	18

Introduction

The IpQuart driver is a Win32 driver model (WDM) device driver for the IP-QuadUART board from Dynamic Engineering. Each IP-QuadUART board has an Exar XR16C854 four channel UART and programmable RS-232/RS-485 Sipex drivers. A separate Device Object controls each IP-QuadUART board, and a separate handle references each Device Object. IO Control calls (IOCTLs) are used to configure the board and ReadFile and WriteFile calls are used to move data in and out of the UART channel data buffers.

Note

This documentation will provide information about all calls made to the driver, and how the driver interacts with the hardware for each of these calls. For more detailed information on the hardware implementation, refer to the IP-QuadUART device user manual (also referred to as the hardware manual).

Driver Installation

Warning: The appropriate IP carrier driver must be installed before any IP modules can be detected by the system.

There are several files provided in each driver package. These files include IpQuart.sys, IpDevice.inf, DDIpQuart.h, IpQuartGUID.h, IpQuartDef.h, IPQTest.exe, and IPQTest source files.

Windows 2000 Installation

Copy IpDevice.inf and IpQuart.sys to a floppy disk, or CD if preferred.

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



Windows XP Installation

Copy IpDevice.inf to the WINDOWS\INF folder and copy IpQuart.sys to a floppy disk, or CD if preferred. Right click on the IpDevice.inf file icon in the WINDOWS\INF folder and select **Install** from the pop-up menu. This will create a precompiled information file (.pnf) in the same directory.

With the hardware installed, power-on the PCI host computer and wait for the **Found New Hardware Wizard** dialogue window to appear. The **IP-QuadUART** should be named in the dialogue box. Follow the steps below:

- Insert the disk prepared above in the appropriate drive.
- Select **Install from a list or specific location**
- Select **Next**
- Select **Don't search. I will choose the driver to install**
- Select **Next**
- Select **Show all devices** from the list
- Select **Next**
- Select **Dynamic Engineering** from the Manufacturer list
- Select **IP-QuadUART Device** from the Model list
- Select **Next**
- Select **Yes** on the Update Driver Warning dialogue box.
- Enter the drive e.g. **A:** in the **Files Needed** dialogue box.
- Select **OK**.
- Select **Finish** to close the **Found New Hardware Wizard**.

This process must be completed for each new device that is installed.

The DDIpQuart.h file is the C header file that defines the Application Program Interface (API) to the driver. The IpQuartGUID.h file is a C header file that defines the device interface identifier for the IpQuart. These files are required at compile time by any application that wishes to interface with the IpQuart driver. The IpQuartDef.h file contains the relevant bit defines for the IP-QuadUART registers. These files are not needed for driver installation.

The IPQTest.exe file is a sample Win32 console application that makes calls into the IpQuart driver to test the driver calls without actually writing an application. It is not required during the driver installation. Open a command prompt console window and type **IPQTest -d0 -?** to display a list of commands (the IPQTest.exe file must be in the directory that the window is referencing). The commands are all of the form **IPQTest -dn -im** where **n** and **m** are the device number and driver ioctl number respectively. This application is intended to test the proper functioning of the driver calls, not for normal hardware operation.



Driver Startup

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

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

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

Below is example code for opening a handle for device *N*, where *N* is the instance number of the device (starting with 0). The device number is underlined and italicized in the SetupDiEnumDeviceInterfaces call.

```
// The maximum length of the device name for
// a given instance of an interface
#define MAX_DEVICE_NAME 256
// Handle to the device object
HANDLE hIpQuart = INVALID_HANDLE_VALUE;
// Return status from command
LONG status;
// Handle to device interface information structure
HDEVINFO hDeviceInfo;
// The actual symbolic link name to use in the createfile
CHAR deviceName[MAX_DEVICE_NAME];
// Size of buffer required to get the symbolic link name
DWORD requiredSize;
// Interface data structures for this device
SP_DEVICE_INTERFACE_DATA interfaceData;
PSP_DEVICE_INTERFACE_DETAIL_DATA pDeviceDetail;

hDeviceInfo = SetupDiGetClassDevs((LPGUID)&GUID_DEVINTERFACE_IPQUART,
                                NULL,
                                NULL,
                                DIGCF_PRESENT | DIGCF_DEVICEINTERFACE);

if(hDeviceInfo == INVALID_HANDLE_VALUE)
{
    printf("***Error: couldn't get class info, (%d)\n",
           GetLastError());
    exit(-1);
}

interfaceData.cbSize = sizeof(interfaceData);

// Find the interface for device N
if(!SetupDiEnumDeviceInterfaces(hDeviceInfo,
                                NULL,
                                (LPGUID)&GUID_DEVINTERFACE_IPQUART,
                                N,
                                &interfaceData))
{
```



```

status = GetLastError();
if(status == ERROR_NO_MORE_ITEMS)
{
    printf("***Error: couldn't find device(no more items), (%d)\n", N);
    SetupDiDestroyDeviceInfoList(hDeviceInfo);
    exit(-1);
}
else
{
    printf("***Error: couldn't enum device, (%d)\n",
        status);
    SetupDiDestroyDeviceInfoList(hDeviceInfo);
    exit(-1);
}
}

// Get the details data to obtain the symbolic link name
if(!SetupDiGetDeviceInterfaceDetail(hDeviceInfo,
    &interfaceData,
    NULL,
    0,
    &requiredSize,
    NULL))
{
    if(GetLastError() != ERROR_INSUFFICIENT_BUFFER)
    {
        printf("***Error: couldn't get interface detail, (%d)\n",
            GetLastError());
        SetupDiDestroyDeviceInfoList(hDeviceInfo);
        exit(-1);
    }
}

// Allocate a buffer to get detail
pDeviceDetail = (PSP_DEVICE_INTERFACE_DETAIL_DATA)malloc(requiredSize);
if(pDeviceDetail == NULL)
{
    printf("***Error: couldn't allocate interface detail\n");
    SetupDiDestroyDeviceInfoList(hDeviceInfo);
    exit(-1);
}
pDeviceDetail->cbSize = sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);

// Get the detail info
if(!SetupDiGetDeviceInterfaceDetail(hDeviceInfo,
    &interfaceData,
    pDeviceDetail,
    requiredSize,
    NULL,
    NULL))
{
    printf("***Error: couldn't get interface detail(2), (%d)\n",
        GetLastError());
    SetupDiDestroyDeviceInfoList(hDeviceInfo);
    free(pDeviceDetail);
    exit(-1);
}
}

```



```

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

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

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

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

```

IO Controls

The driver uses IO Control calls (IOCTLs) to configure the device. IOCTLs refer to a single Device Object in the driver, 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.

IOCTL_IPQUART_GET_INFO

Function: Returns the current driver version, instance number, UART device ID and revision, and Xilinx revision.

Input: None

Output: DRIVER_IPQUART_DEVICE_INFO structure

Notes: This call does not access the hardware, only driver parameters. See DDIpQuart.h for the definition of DRIVER_IPQUART_DEVICE_INFO.

IOCTL_IPQUART_SET_ACTIVE_CHAN

Function: Determines which UART channel will be the target of the next ReadFile or WriteFile call.

Input: Channel number (unsigned character)

Output: None

Notes: This call does not access the hardware, only driver parameters.



IOCTL_IPQUART_SET_IP_CONTROL

Function: Sets the configuration of the IP slot.

Input: Register configuration (unsigned long integer)

Output: None

Notes: Controls the IP clock speed and interrupt enables for the IP slot that the board occupies. See the bit definitions in the IpQuartDef.h header file for more information.

IOCTL_IPQUART_GET_IP_CONTROL

Function: Returns the configuration of the IP slot.

Input: None

Output: Register configuration (unsigned long integer)

Notes: Returns the slot configuration register value for the IP slot that the board occupies. See the bit definitions in the IpQuartDef.h header file for more information.

IOCTL_IPQUART_SET_BASE_CONFIG

Function: Sets configuration parameters in the IP-QuadUART base control register.

Input: IPQUART_BASE_CONFIG structure

Output: None

Notes: Selects the reference clock source(s) for the UART channels and the bus timeout interrupt enable state. See DDIpQuart.h for the definition of IPQUART_BASE_CONFIG.

IOCTL_IPQUART_GET_BASE_CONFIG

Function: Returns the configuration of the base control register.

Input: None

Output: IPQUART_BASE_CONFIG structure

Notes: Returns the values set in the previous call.

IOCTL_IPQUART_SET_CHAN_CONFIG

Function: Sets configuration parameters in an IP-QuadUART channel control register.

Input: IPQUART_CHAN_CONFIG structure

Output: None

Notes: Controls the UART channel interrupt enable, the I/O driver configuration, and the IP bus to UART data interface configuration. See DDIpQuart.h for the definition of IPQUART_CHAN_CONFIG.



IOCTL_IPQUART_GET_CHAN_CONFIG

Function: Returns the configuration of an IP-QuadUART channel control register.

Input: Channel number (unsigned character)

Output: IPQUART_CHAN_CONFIG structure

Notes: Returns the values set in the previous call.

IOCTL_IPQUART_SET_UART_DATA_CONFIG

Function: Sets the configuration of a UART channel data word and baud rate.

Input: UART_DATA_CONFIG structure

Output: None

Notes: Controls the baud rate, number of data bits, number of stop bits and the parity configuration of the specified channel. Accesses the UART LCR, DLL, and DLM registers. See DDlpQuart.h for the definition of UART_DATA_CONFIG. See the XR16C854 data sheet for the UART internal register descriptions.

IOCTL_IPQUART_GET_UART_DATA_CONFIG

Function: Returns the data configuration of a UART channel.

Input: Channel number (unsigned character)

Output: UART_DATA_CONFIG structure

Notes: Returns the values set in the previous call.

IOCTL_IPQUART_SET_UART_INTEN

Function: Sets the possible interrupt sources for a UART channel.

Input: UART_INT_CONFIG structure

Output: None

Notes: Selects any of seven interrupt sources for a UART channel. Accesses the UART IER register. See DDlpQuart.h for the definition of UART_INT_CONFIG. See the XR16C854 data sheet for the UART interrupt enable register description.

IOCTL_IPQUART_GET_UART_INTEN

Function: Returns the interrupt enable configuration of a UART channel.

Input: Channel number (unsigned character)

Output: UART_INT_CONFIG structure

Notes: Returns the values set in the previous call.



IOCTL_IPQUART_SET_UART_MODEM_CONTROL

Function: Sets the modem control signals and internal loop-back enable for a UART channel.

Input: UART_MODEM_CONTROL structure

Output: None

Notes: Controls the state of the modem control signals (RTS, DTR) and internal loop-back signals (OP1, OP2) for a UART channel. Accesses the UART MCR register. See DDIpQuart.h for the definition of UART_MODEM_CONTROL. See the XR16C854 data sheet for the UART modem control register description.

IOCTL_IPQUART_GET_UART_MODEM_CONTROL

Function: Returns the modem control signals for a UART channel.

Input: Channel number (unsigned character)

Output: UART_MODEM_CONTROL structure

Notes: Returns the values set in the previous call.

IOCTL_IPQUART_SET_UART_FLOW_CONTROL_PARAMS

Function: Sets the flow control parameters for a UART channel.

Input: UART_FLOW_PARAMS structure

Output: None

Notes: Sets the Rx and Tx FIFO trigger levels, Rx hysteresis value, and the Xon and Xoff character values. Accesses the UART FCTR, EMSR, TRG, XON1, XON2, XOFF1, and XOFF2 registers. See DDIpQuart.h for the definition of UART_FLOW_PARAMS. See the XR16C854 data sheet for the UART internal register descriptions. The EMSR and TRG registers are write only, so there is no corresponding GET_UART_FLOW_CONTROL_PARAMS for this call.

IOCTL_IPQUART_SET_UART_FLOW_CONTROL_MODE

Function: Sets the flow control mode for a UART channel.

Input: UART_FLOW_CONFIG structure

Output: None

Notes: Controls whether hardware, software, or no flow control is used for a UART channel, and further details of the selected mode. Accesses the UART EFR register. See DDIpQuart.h for the definition of UART_FLOW_CONFIG. See the XR16C854 data sheet for the UART enhanced function register description.



IOCTL_IPQUART_GET_UART_FLOW_CONTROL_MODE

Function: Returns the flow control mode for a UART channel.

Input: Channel number (unsigned character)

Output: UART_FLOW_CONFIG structure

Notes: Returns the values set in the previous call.

IOCTL_IPQUART_WRITE_UART_DATA_BYTE

Function: Writes one byte of data to a UART channel.

Input: UART_WRITE_BYTE structure

Output: None

Notes: The UART_WRITE_BYTE structure has two fields, channel and data. This call is used when the Tx FIFO for the channel is disabled or to write a small amount of data to a UART Tx channel. See DDIpQuart.h for the definition of UART_WRITE_BYTE.

IOCTL_IPQUART_READ_UART_DATA_BYTE

Function: Reads a single byte of data from a UART channel.

Input: Channel number (unsigned character)

Output: Data byte (unsigned character)

Notes: This call is used when the Rx FIFO for the channel is disabled or to read a small amount of data from a UART Rx channel.

IOCTL_IPQUART_SET_TIMEOUT_CONFIG

Function: Sets the bus timeout count and data values.

Input: IPQUART_TIMEOUT_CONFIG structure

Output: None

Notes: Sets the timeout count, the number of IP clocks that the IP bus interface will wait before signaling a timeout interrupt and returning the data specified in the timeout data field. This will only occur when pre-read data is accessed and there is insufficient data stored to satisfy the request. See DDIpQuart.h for the definition of IPQUART_TIMEOUT_CONFIG.

IOCTL_IPQUART_GET_TIMEOUT_CONFIG

Function: Returns the bus timeout count and data values.

Input: None

Output: IPQUART_TIMEOUT_CONFIG structure

Notes: Returns the values set in the previous call.



IOCTL_IPQUART_RESET_UART

Function: Resets the UART device.

Input: None

Output: None

Notes: Resets the UART and restores the baud rate, data word configuration, interrupt enables, and FIFO enables that were in effect before the reset. All other values are returned to the defaults.

IOCTL_IPQUART_CONFIGURE_UART_FIFOS

Function: Enables and/or resets any or all of the UART FIFOs.

Input: IPQUART_FIFO_CONFIG structure

Output: None

Notes: Controls whether the UART FIFOs are enabled or disabled. This is selectable on a per channel basis. If the FIFOs are enabled for a channel, either the transmit or receive FIFOs can be reset. See DDIpQuart.h for the definition of IPQUART_FIFO_CONFIG. If the Xilinx is below revision C, an Rx FIFO reset will not delete any data pre-read from the Rx FIFO.

IOCTL_IPQUART_GET_UART_STATUS

Function: Reads various status values for a UART channel.

Input: Channel number (unsigned character)

Output: UART_STATUS structure

Notes: Reads and returns the value of the UART interrupt status register, line status register, and modem status register as well as the Rx and Tx FIFO data counts. See DDIpQuart.h for the definition of UART_STATUS.

IOCTL_IPQUART_GET_STATUS

Function: Returns the status bits in the INT_STAT register.

Input: None

Output: Unsigned short integer

Notes: Reads and returns the value of the INT_STAT register which indicates the interrupt source. See the bit definitions in the IpQuartDef.h header file for more information.



IOCTL_IPQUART_REGISTER_EVENT

Function: Registers an event to be signaled when an interrupt occurs.

Input: Handle to 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 an interrupt is serviced. The user interrupt service routine waits on this event, allowing it to respond to the interrupt. In order to un-register the event, set the event handle to NULL while making this call.

IOCTL_IPQUART_ENABLE_INTERRUPT

Function: Enables the master interrupt.

Input: None

Output: None

Notes: Sets the master interrupt enable, leaving all other bit values in the IP-QuadUART base control register unchanged. Also checks the state of the IP slot control register interrupt 0 enable bit in the saved configuration, and sets it if needed. This IOCTL is used in the user interrupt processing function to re-enable the interrupts after they were disabled in the driver interrupt service routine. This call allows that function to enable the interrupts without knowing the particulars of the other configuration bits.

IOCTL_IPQUART_DISABLE_INTERRUPT

Function: Disables the master interrupt.

Input: None

Output: None

Notes: Clears the master interrupt enable, leaving all other bit values in the IP-QuadUART base control register unchanged. This IOCTL is used when interrupt processing is no longer desired.

IOCTL_IPQUART_FORCE_INTERRUPT

Function: Causes a system interrupt to occur.

Input: None

Output: None

Notes: Causes an interrupt to be asserted on the IP bus. This IOCTL is used for development, to test interrupt processing.



IOCTL_IPQUART_SET_VECTOR

Function: Sets the value of the interrupt vector.

Input: Unsigned character

Output: None

Notes: This value will be driven onto the low byte of the data bus in response to an INT_SEL strobe, which is used in vectored interrupt cycles. This value will be read in the interrupt service routine and stored for future reference.

IOCTL_IPQUART_GET_VECTOR

Function: Returns the current interrupt vector value.

Input: None

Output: Unsigned character

Notes: Returns the values set in the previous call.

IOCTL_IPQUART_GET_ISR_STATUS

Function: Returns the interrupt status, vector, and UART interrupt status register values read in the last ISR.

Input: None

Output: IPQUART_INT_STAT structure

Notes: The status contains the contents of the INT_STAT register read in the last driver interrupt service routine execution. If bit 12 is set, it indicates that a bus error occurred for this IP slot. The interrupt vector and the interrupt status register values for any UART channels that had interrupts pending in the last ISR are also returned. See DDIpQuart.h for the definition of IPQUART_INT_STAT.



Write

Transmit data can be written to the transmit FIFO of the current active channel using a WriteFile() call. The user supplies the device handle, a pointer to the buffer containing the data, the number of bytes to write, a pointer to a variable to store the amount of data actually transferred, and a pointer to an optional Overlapped structure for performing asynchronous IO. The number of bytes requested and the current FIFO data level are checked to see how much data can be sent. If the FIFO is disabled, at most one byte can be sent. Otherwise, the command is executed with successive writes to the Tx FIFO port. If 16-bit writes are enabled, the driver takes advantage of the carrier 32-bit double-write capability to load four FIFO words with a single PCI write until less than four bytes remain to be sent. See Win32 help files for details of the WriteFile() call.

Read

Received data can be read from the receive FIFO of the current active channel using a ReadFile() call. The user supplies the device handle, a pointer to the buffer in which to store the data, the number of bytes to read, a pointer to a variable to store the amount of data actually transferred, and a pointer to an optional Overlapped structure for performing asynchronous IO. The number of bytes requested and the current FIFO data level are checked to see how much data can be read. If the FIFO is disabled, at most one byte can be read. Otherwise, the command is executed with successive reads from the Rx FIFO port. If 16-bit reads are enabled, the driver takes advantage of the carrier 32-bit double-read capability to read four FIFO words with a single PCI read until less than four bytes remain to be read. See Win32 help files for the details of the ReadFile() call.

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 a cockpit error rather than an error with the driver. When you are sure or at least willing to pay to have someone help then call the Customer Service Department and arrange to speak with an engineer. We will work with you to determine the cause of the issue. If the issue is one of a defective driver we will correct the problem and provide an updated module(s) to you [no cost]. If the issue is of the customer's making [anything that is not the driver] the engineering time will be invoiced to the customer. Pre-approval may be required in some cases depending on the customer's invoicing policy.

Out of Warranty Repairs

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

For Service Contact:

Customer Service Department
Dynamic Engineering
435 Park Dr.
Ben Lomond, CA 95005
831-336-8891
831-336-3840 fax
support@dyneng.com

All information provided is Copyright Dynamic Engineering

