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

IP-BISERIAL6-SIB

Driver Documentation

Developed with Windows Driver Foundation Ver1.9

Manual Revision A
Corresponding Hardware: Revision A
10-2016-3201
FLASH revision A1

IP-BIS6-SIB

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

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.

Table of Contents

Introduction

The IP-BISERIAL6-SIB driver is a Windows device driver for the IP-Test Industry-pack (IP) module from Dynamic Engineering. This driver was developed with the Windows Driver Foundation version 1.9 (WDF) from Microsoft, specifically the Kernel-Mode Driver Framework (KMDF).

The IP-BISERIAL6-SIB driver package has two parts. The driver is installed into the Windows® OS, and the User Application "UserApp" executable.

The driver is delivered as installed or executable items to be used directly or indirectly by the user. The UserApp code is delivered in source form [C] and is for the purpose of providing a reference to using the driver.

UserApp is a stand-alone code set with a simple, and powerful menu plus a series of "tests" that can be run on the installed hardware. Each of the tests execute calls to the driver, pass parameters and structures, and get results back. With the sequence of calls demonstrated, the functions of the hardware are utilized for loop-back testing. The software is used for manufacturing test at Dynamic Engineering.

The test software can be ported to your application to provide a running start. It is recommended to port the ChanPreloadReg test to your application to get started. The test is simple and will quickly demonstrate the end-to-end operation of your application making calls to the driver and interacting with the hardware.

The menu allows the user to add tests, to run sequences of tests, to run until a failure occurs and stop or to continue, to program a set number of loops to execute and more. The user can add tests to the provided test suite to try out application ideas before committing to your system configuration. In many cases the test configuration will allow faster debugging in a more controlled environment before integrating with the rest of the system. The test suite is designed to accommodate up to 5 boards. The number of boards can be expanded. See Main.c to increase the number of handles.

The hardware manual defines the pinout, the bitmaps and detailed configurations for each feature of the design. The driver handles all aspects of interacting with the hardware. For added explanations about what some of the driver functions do, please refer to the hardware manual.

We strive to make a useable product, and while we can guarantee operation we can't foresee all concepts for client implementation. If you have suggestions for extended features, special calls for particular set-ups or whatever please share them with us, [engineering@dyneng.com] and we will consider and in many cases add them.

IP-BISERIAL6-SIB has a Spartan2 Xilinx FPGA to implement the IP Interface, protocol

control and status for the IO.

When the IP-BISERIAL6-SIB board is recognized by the IP Carrier Driver, the carrier driver will start the IP-BISERIAL6-SIB driver which will create a device object for the board. If more than one is found additional copies of the driver are loaded. The carrier driver will load the info storage register on the IP-BISERIAL6-SIB with the carrier switch setting and the slot number of the IP-BISERIAL6-SIB device. From within the IP-BISERIAL6-SIB driver the user can access the switch and slot information to determine the specific device being accessed when more than one are installed.

The reference software application has a loop to check for devices. The number of devices found, the locations, and device count are printed out at the top of the menu.

IO Control calls (IOCTLs) are used to configure the board and read status. Read and Write calls are used to move 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 IP-BISERIAL6-SIB user manual (also referred to as the hardware manual).

Driver Installation

There are several files provided in each driver package. These files include IpBis6Sib.sys, IpBis6SibPublic.h, IpPublic.h, WdfCoInstaller01009.dll, IpDevices.inf and IpDevices.cat.

IpBis6SibPublic.h and IpPublic.h are C header files that define the Application Program Interface (API) to the driver. These files are required at compile time by any application that wishes to interface with the driver, but are not needed for driver installation.

Note: Other IP module drivers are included in the package since they were all signed together and must be present to validate the digital signature. These other IP module driver files must be present when the IpBis6Sib driver is installed, to verify the digital signature in IpDevices.cat, otherwise they can be ignored.

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

Windows 7 Installation

Copy IpDevices.inf, IpDevices.cat, WdfCoInstaller01009.dll, IpBis6Sib.sys and the other IP module drivers to a removable memory device or other accessible location as preferred.

With the IP hardware installed, power-on the host computer.

- Open the ***Device Manager*** from the control panel.
- Under ***Other devices*** there should be an item for each IP module installed on the IP carrier. The label for a module installed in the first slot of the first PCIe3IP carrier would read ***PcieCar0 IP Slot A****.
- Right-click on the first device and select ***Update Driver Software***.
- Insert the removable memory device prepared above if necessary.
- Select ***Browse my computer for driver software***.
- Select ***Browse*** and navigate to the memory device or other location prepared above.
- Select ***Next***. The IpBis6Sib device driver should now be installed.
- Select ***Close*** to close the update window.
- Right-click on the remaining IP slot icons and repeat the above procedure as necessary.

***** If the [***Carrier] IP Slot [x]*** devices are not displayed, click on the ***Scan for hardware changes*** icon on the Device Manager tool-bar.

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 IpBis6SibPublic.h.

The *main.c* file provided with the user test software can be used as an example to show how to obtain a handle to an IpBis6Sib device.

IO Controls

The driver uses IO Control calls (IOCTLs) to configure the device. IOCTLs refer to a single Device Object, which controls a single module. 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 IpBis6Sib driver are described below:

IOCTL_IP_BIS6_SIB_GET_INFO

Function: Returns the driver and firmware revisions, module instance number and location and other information.
Input: None
Output: DRIVER_IP_DEVICE_INFO structure
Notes: This call does not access the hardware, only stored driver parameters. NewIpCntl indicates that the module's carrier has expanded slot control capabilities. See the definition of DRIVER_IP_DEVICE_INFO below.

```
 //  Driver version and instance/slot information
typedef struct _DRIVER_IP_DEVICE_INFO {
   UCHAR    DriverRev;         // Driver revision
   UCHAR    FirmwareRev;       // Firmware major revision
   UCHAR    FirmwareRevMin;    // Firmware minor revision
   UCHAR    InstanceNum;       // Zero-based device number
   UCHAR    CarrierSwitch;     // 0..0xFF
   UCHAR    CarrierSlotNum;    // 0..7 -> IP slots A, B, C, D, E, F, G or H
   UCHAR    CarDriverRev;      // Carrier driver revision
   UCHAR    CarFirmwareRev;    // Carrier firmware major revision
   UCHAR    CarFirmwareRevMin;// Carrier firmware minor revision
   UCHAR    CarCPLDRev;        //**Used for PCIe carriers only**0xFF for others
   UCHAR    CarCPLDRevMin;     //**Used for PCIe carriers only**0xFF for others
   BOOLEAN  Ip32MCapable;      // IP capable of both 8MHz and 32MHz operation
   BOOLEAN  NewIpCntl;         // New IP slot control interface
   WCHAR    LocationString[IP_LOC_STRING_SIZE];
} DRIVER_IP_DEVICE_INFO, *PDRIVER_IP_DEVICE_INFO;
```

IOCTL_IP_BIS6_SIB_SET_IP_CONTROL

Function: Sets various control parameters for the IP slot the module is installed in.
Input: IP_SLOT_CONTROL structure
Output: None
Notes: Controls the IP clock speed, interrupt enables and data manipulation options for the IP slot that the board occupies. See the definition of IP_SLOT_CONTROL below. For more information refer to the IP carrier hardware manual.

```
typedef struct _IP_SLOT_CONTROL {
   BOOLEAN  Clock32Sel;
   BOOLEAN  ClockDis;
   BOOLEAN  ByteSwap;
   BOOLEAN  WordSwap;
   BOOLEAN  WrIncDis;
   BOOLEAN  RdIncDis;
   UCHAR    WrWordSel;
   UCHAR    RdWordSel;
   BOOLEAN  BsErrTmOutSel;
   BOOLEAN  ActCountEn;
} IP_SLOT_CONTROL, *PIP_SLOT_CONTROL;
```

IOCTL_IP_BIS6_SIB_GET_IP_STATE

Function: Returns control/status information for the IP slot the module is installed in.
Input: None
Output: IP_SLOT_STATE structure
Notes: Returns the slot control parameters set in the previous call as well as status information for the IP slot that the board occupies. See the definition of IP_SLOT_STATE below.

```
typedef struct _IP_SLOT_STATE {
   BOOLEAN  Clock32Sel;
   BOOLEAN  ClockDis;
   BOOLEAN  ByteSwap;
   BOOLEAN  WordSwap;
   BOOLEAN  WrIncDis;
   BOOLEAN  RdIncDis;
   UCHAR    WrWordSel;
   UCHAR    RdWordSel;
   BOOLEAN  BsErrTmOutSel;
   BOOLEAN  ActCountEn;
 // Slot Status
   BOOLEAN  IpInt0En;
   BOOLEAN  IpInt1En;
   BOOLEAN  IpBusErrIntEn;
   BOOLEAN  IpInt0Actv;
   BOOLEAN  IpInt1Actv;
   BOOLEAN  IpBusError;
   BOOLEAN  IpForceInt;
   BOOLEAN  WrBusError;
   BOOLEAN  RdBusError;
} IP_SLOT_STATE, *PIP_SLOT_STATE;
```

IOCTL_IP_BIS6_SIB_SET_BASE_CONFIG

Function: Sets configuration parameters in the base control register.
Input: IP_BIS6_SIB_BASE0_CONFIG structure
Output: None
Notes: See the definition of IP_BIS6_SIB_BASE_CONFIG below. Bit definitions can be found under the 'IPBISVI_BASE' section under Register Definitions in the Hardware manual.

```
typedef struct _IP_BIS6_SIB_BASE_CONFIG {
   BOOLEAN     Clock32;
   BOOLEAN     MasterChanEn;
   BOOLEAN     SdClkEn;
} IP_BIS6_SIB_BASE_CONFIG, *PIP_BIS6_SIB_BASE_CONFIG;
```

IOCTL_IP_BIS6_SIB_GET_BASE_CONFIG

Function: Returns the configuration of the base control register.
Input: None
Output: IP_BIS6_SIB_BASE_CONFIG structure
Notes: Returns the values set in the previous call.

IOCTL_IP_BIS6_SIB_GET_IP_ID

Function: Returns IP module information.
Input: None
Output: IP-IDENTITY structure
Notes: See the definition of I IP_IDENTITY below.

```
typedef struct _IP_IDENTITY {
   UCHAR    IpManuf;
   UCHAR    IpModel;
   UCHAR    IpRevision;
   UCHAR    IpCustomer;
   USHORT   IpVersion;
} IP_IDENTITY, *PIP_IDENTITY;
```

IOCTL_IP_BIS6_SIB_GET_VERSION

Function: Returns the Module driver flash minor and major revisions.
Input: None
Output: IP_MOD_VERSION structure
Notes: See the definition of I IP_MOD_VERSION below. Bit definitions can be found under the 'IPBISVI_REV' section under Register Definitions in the Hardware manual.

```
typedef struct _IP_MOD_VERSION {
   UCHAR    minorFlashRev;
   UCHAR    majorFlashRev;
} IP_MOD_VERSION, *PIP_MOD_VERSION;
```

IOCTL_IP_BIS6_SIB_SET_GATE_COUNT

Function: Writes a 16 bit value.
Input: ULONG
Output: None
Notes: Write to set the register check count. Detailed definition can be found in the 'IPBISVI_SIB_GATECNT' section under Register Definitions in the Hardware manual.

IOCTL_IP_BIS6_SIB_GET_GATE_COUNT

Function: Returns a bit value.
Input: None
Output: ULONG
Notes: Read register for the specified channel

IOCTL_IP_BIS6_SIB_GET_ISR_STATUS

Function: Interrupt status, and vector read in the ISR from the last user interrupt.
Input: None
Output: IP-BIS6-SIB_ISR_STAT
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. Bit definitions for either of the channel status can be found under the 'IPBISVI_CHAN_STAT' section under Register Definitions in the Hardware manual.

```
typedef struct _IP_BIS6_SIB_ISR_STAT {
   USHORT   InterruptStatus;  // stored Interrupt status from ISR
   USHORT   InterruptVector;  // stored Interrupt vector from ISR
   USHORT   Chan0Int;
   USHORT   Chan1Int;
} IP_BIS6_SIB_ISR_STAT, *PIP_BIS6_SIB_ISR_STAT;
```

IOCTL_IP_BIS6_SIB_GET_BASE_STATUS

Function: Returns the current base interrupt status.
Input: None
Output: PUSHORT
Notes:

IOCTL_IP_BIS6_SIB_CLEAR_BASE_STATUS

Function: Clears the latched interrupt status bits.
Input: None
Output: PUSHORT
Notes: Write a 1 to clear the latched bit.

IOCTL_IP_BIS6_SIB_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_IP_BIS6_SIB_ENABLE_BASE_INT

Function: Sets the master interrupt enable.
Input: None
Output: None
Notes: Sets the master interrupt enable, leaving all other bit values in the base register unchanged. This IOCTL is used in the user interrupt processing function to re-enable the interrupts after they were disabled in the driver ISR. This allows the driver to set the master interrupt enable without knowing the state of the other base configuration bits.

IOCTL_IP_BIS6_SIB_DISABLE_BASE_INT

Function: Clears the master interrupt enable.
Input: None
Output: None
Notes: Clears the master interrupt enable, leaving all other bit values in the base register unchanged. This IOCTL is used when interrupt processing is no longer desired.

IOCTL_IP_BIS6_SIB_FORCE_BASE_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_IP_BIS6_SIB_SET_VECTOR

Function: Writes an 8 bit value to the interrupt vector register.
Input: UCHAR
Output: None
Notes: Required when used in non auto-vectored systems.

IOCTL_IP_BIS6_SIB_GET_VECTOR

Function: Returns a stored vector value.
Input: None
Output: UCHAR
Notes:

IOCTL_IP_BIS6_SIB_SET_CHAN_CONFIG

Function: Sets configuration parameters in the channel control register.
Input: IP_BIS6_SIB_CHAN_CONFIG structure
Output: None
Notes: See the definition of IP-BIS6-SIB_CHAN_CONFIG below. Bit definitions can be found under 'IPBISVI_SIB_CHAN' section under Register Definitions in the Hardware manual.

```
typedef struct _IP_BIS6_SIB_CHAN_CONFIG {
   UCHAR       Chan;
   MODE_SEL    ModeSel;
   BOOLEAN     SwRest;
   BOOLEAN     ClearCount;
   BOOLEAN     CtsDisable;
   BOOLEAN     CtsSense;
   BOOLEAN     UseBaseEn;
   BOOLEAN     EnChan;
} IP_BIS6_SIB_CHAN_CONFIG, *PIP_BIS6_SIB_CHAN_CONFIG;
```

IOCTL_IP_BIS6_SIB_GET_CHAN_CONFIG

Function: Returns the configuration of the channel control register.
Input: None
Output: IP_BIS6_SIB_CHAN_CONFIG structure
Notes: Returns the values set in the previous call.

IOCTL_IP_BIS6_SIB_GET_CHAN_STATUS

Function: Returns the current channel interrupt status for specified channel.
Input: Chan (UCHAR)
Output: IP_BIS6_SIB_CHAN_STAT
Notes: Bit definitions can be found under 'IPBISVI_SIB_STAT' section under Register Definitions in the Hardware manual.

```
typedef struct _IP_BIS6_SIB_CHAN_STAT {
   UCHAR       Chan;
   USHORT      Status;
} IP_BIS6_SIB_CHAN_STAT, *PIP_BIS6_SIB_CHAN_STAT;
```

IOCTL_IP_BIS6_SIB_CLEAR_CHAN_STATUS

Function: Clear the interrupt status. And returns the current channel interrupt status for specified channel
Input: IP_BIS6_SIB_CHAN_STAT
Output: USHORT
Notes: Bit definitions can be found under 'IPBISVI_SIB_STAT' section under Register Definitions in the Hardware manual.

IOCTL_IP_BIS6_SIB_SET_ALMOST_FULL

Function: Writes a 16 bit value.
Input: IP_BIS6_SIB_ALMOST_FULL
Output: None
Notes: Write to set the register almost full for specified channel. Detailed definition can be found in the 'IPBISVI_SIB_AlmostFull' section under Register Definitions in the Hardware manual.

```
typedef struct _IP_BIS6_SIB_AFL {
   UCHAR    Chan;
   USHORT   AlmostFull;
} IP_BIS6_SIB_AFL, *PIP_BIS6_SIB_AFL;
```

IOCTL_IP_BIS6_SIB_GET_ALMOST_FULL

Function: Returns a 16 bit value.
Input: UCHAR
Output: IP_BIS6_SIB_ALMOST_FULL
Notes: Read register for the specified channel

IOCTL_IP_BIS6_SIB_SET_ALMOST_EMPTY

Function: Writes a 16 bit value.
Input: IP_BIS6_SIB_ALMOST_EMPTY
Output: None
Notes: Write to set the register almost empty for specified channel. Detailed definition can be found in the 'IPBISVI_SIB_AlmostEmpty' section under Register Definitions in the Hardware manual.

```
typedef struct _IP_BIS6_SIB_AMT {
   UCHAR    Chan;
   USHORT   AlmostEmpty;
} IP_BIS6_SIB_AMT, *PIP_BIS6_SIB_AMT;
```

IOCTL_IP_BIS6_SIB_GET_ALMOST_EMPTY

Function: Returns a 16 bit value.
Input: UCHAR
Output: IP_BIS6_SIB_ALMOST_EMPTY
Notes: Read register for the specified channel

IOCTL_IP_BIS6_SIB_SET_TX_DATA

Function: Writes a 16 bit value.
Input: IP_BIS6_SIB_DATA
Output: None
Notes: Detailed definition can be found in the 'IPBISVI_SIB_FIFO' section under Register Definitions in the Hardware manual.

```
typedef struct _IP_BIS6_SIB_DATA {
   UCHAR    Chan;
   USHORT   Data;
} IP_BIS6_SIB_DATA, *PIP_BIS6_SIB_DATA;
```

IOCTL_IP_BIS6_SIB_GET_RX_DATA

Function: Returns a 16 bit value.
Input: None
Output: IP_BIS6_SIB_DATA
Notes: Detailed definition can be found in the 'IPBISVI_SIB_FIFO' section under Register Definitions in the Hardware manual.

IOCTL_IP_BIS6_SIB_GET_CHECK_CNT

Function: Returns a 16 bit count value.
Input: UCHAR
Output: USHORT
Notes: Read check count register for the specified channel. Detailed definition can be found in the 'IPBISVI_SIB_Check_count' section under Register Definitions in the Hardware manual.

IOCTL_IP_BIS6_SIB_SET_CHAN_INT

Function: Sets the specified channel interrupt enable.
Input: channel UCHAR
Output: None
Notes: Sets the channel interrupt enable, leaving all other bit values in the register unchanged.

IOCTL_IP_BIS6_SIB_GET_CHAN_INT

Function: Clears specified channel interrupt enable.
Input: channel UCHAR
Output: None
Notes: Clears the master interrupt enable, leaving all other bit values in the base register unchanged.

IOCTL_IP_BIS6_SIB_FORCE_CHAN_INT

Function: Causes a system interrupt to occur.
Input: channel UCHAR
Output: None
Notes: Causes an interrupt to be asserted on the IP bus. This IOCTL is used for development, to test interrupt processing.

IOCTL_IP_BIS6_SIB_GET_TX_FIFO_CNT

Function: Returns a 16 bit count value.
Input: UCHAR
Output: USHORT
Notes: Read TX FIFO count register for the specified channel. Detailed definition can be found in the 'IPBISVI_SIB_TxFifoCnt' section under Register Definitions in the Hardware manual.

IOCTL_IP_BIS6_SIB_GET_RX_FIFO_CNT

Function: Returns a 16 bit count value.
Input: UCHAR
Output: USHORT
Notes: Read RX FIFO count register for the specified channel. Detailed definition can be found in the 'IPBISVI_SIB_RxFifoCnt' section under Register Definitions in the Hardware manual.

IOCTL_IP_BIS6_SIB_SET_ACTIVE_WR

Function: Specifies the target channel for WriteFile calls.
Input: Channel number (unsigned character)
Output: None
Notes: This is necessary when using the WriteFile function.

IOCTL_IP_BIS6_SIB_SET_ACTIVE_RD

Function: Specifies the target channel for ReadFile calls.
Input: Channel number (unsigned character)
Output: None
Notes: This is necessary when using the ReadFile function.

Write

IP BIS6 SIB RAM data is written to the device using the write command. Writes are executed using the function WriteFile() and passing in the handle to the device opened with CreateFile(), a pointer to a pre-allocated buffer containing the data to be written, an unsigned long integer that represents the size of that buffer in bytes, a pointer to an unsigned long integer to contain the number of bytes actually written, and a pointer to an optional Overlapped structure for performing asynchronous IO.

Read

IP BIS6 SIB RAM data is read from the device using the read command. Reads are executed using the function ReadFile() and passing in the handle to the device opened with CreateFile(), a pointer to a pre-allocated buffer that will contain the data read, an unsigned long integer that represents the size of that buffer in bytes, a pointer to an unsigned long integer to contain the number of bytes actually read, and a pointer to an optional Overlapped structure for performing asynchronous IO.

Warranty and Repair

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

Dynamic Engineering's warranty of and liability for defective products is limited to that set forth herein. Dynamic Engineering disclaims and excludes all other product warranties and product liability, expressed or implied, including but not limited to any implied warranties of merchantability 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.

Support

The software described in this manual is provided at no cost to clients who have purchased the corresponding hardware. Minimal support is included along with the documentation. For help with integration into your project please contact sales@dyneng.com for a support contract. Several options are available. With a contract in place Dynamic Engineers can help with system debugging, special software development, or whatever you need to get going.

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