DartBase
&
DartChan

Driver Documentation

Win32 Driver Model

Revision A
Corresponding Hardware: Revision A
10-2009-0101
DartBase & DartChan
WDM Device Drivers for the
cPCI-Dart Eight-Channel High-Speed-
Serial Analog/Digital Conversion In
and Out with Re-configurable I/O logic

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.

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

©2009 by Dynamic Engineering
Other trademarks and registered trademarks are
owned by their respective manufactures.
Table of Contents

Introduction ........................................................................................................................................... 4
Note ....................................................................................................................................................... 4
Driver Installation .............................................................................................................................. 4
Windows 2000 Installation ............................................................................................................... 6
Windows XP Installation .................................................................................................................. 6
Driver Startup ...................................................................................................................................... 7
IO Controls .......................................................................................................................................... 13
   IOCTL_DART_BASE_GET_INFO ..................................................................................................... 13
   IOCTL_DART_BASE_SET_CONFIG ............................................................................................... 13
   IOCTL_DART_BASE_GET_CONFIG ............................................................................................... 14
   IOCTL_DART_BASE_GET_STATUS ............................................................................................... 14
   IOCTL_DART_BASE_SET_TARGET_REG ...................................................................................... 14
   IOCTL_DART_BASE_GET_TARGET_REG ...................................................................................... 14
   IOCTL_DART_BASE_REGISTER_EVENT ..................................................................................... 14
   IOCTL_DART_BASE_ENABLE_INTERRUPT .................................................................................... 15
   IOCTL_DART_BASE_DISABLE_INTERRUPT ............................................................................... 15
   IOCTL_DART_BASE_FORCE_INTERRUPT ....................................................................................... 15
   IOCTL_DART_BASE_GET_ISR_STATUS ......................................................................................... 15
   IOCTL_DART_BASE_LOAD_DB ....................................................................................................... 15
   IOCTL_DART_BASE_RESET_DB ..................................................................................................... 16
   IOCTL_DART_CHAN_GET_INFO ................................................................................................. 16
   IOCTL_DART_CHAN_SET_CONFIG ............................................................................................... 16
   IOCTL_DART_CHAN_GET_CONFIG ............................................................................................... 16
   IOCTL_DART_CHAN_GET_STATUS ............................................................................................... 16
   IOCTL_DART_CHAN_SET_FIFO_LEVELS ....................................................................................... 17
   IOCTL_DART_CHAN_GET_FIFO_LEVELS ....................................................................................... 17
   IOCTL_DART_CHAN_GET_FIFO_COUNTS ...................................................................................... 17
   IOCTL_DART_CHAN_RESET_FIFO ................................................................................................. 17
   IOCTL_DART_CHAN_WRITE_FIFO ................................................................................................. 17
   IOCTL_DART_CHAN_READ_FIFO ................................................................................................. 17
   IOCTL_DART_CHAN_REGISTER_EVENT ...................................................................................... 18
   IOCTL_DART_CHAN_ENABLE_INTERRUPT ...................................................................................... 18
   IOCTL_DART_CHAN_DISABLE_INTERRUPT .................................................................................... 18
   IOCTL_DART_CHAN_FORCE_INTERRUPT ....................................................................................... 18
   IOCTL_DART_CHAN_GET_ISR_STATUS ......................................................................................... 19
Write .................................................................................................................................................... 19
Read .................................................................................................................................................... 19
Warranty and Repair .......................................................................................................................... 20
Service Policy ...................................................................................................................................... 20
   Out of Warranty Repairs ............................................................................................................. 20
For Service Contact ........................................................................................................................... 20
Introduction
The DartBase and DartChan drivers are Win32 driver model (WDM) device drivers for the cPCI-Dart from Dynamic Engineering.

The cPCI-Dart module consists of two boards, each featuring a Spartan3-2000 Xilinx FPGA to implement the PCI interface, FIFOs and protocol control, status and data I/O for eight Dart channels. Each channel has four 1k by 32-bit data FIFOs, two on the base-board and two on the daughter-board, to handle data transfers in each direction.

The DartChan driver WriteFile() call will initiate a DMA transfer from host memory to the PCI bus to the base-board transmit FIFO, a high-speed-serial data-link and into the daughter-board transmit FIFO and the ReadFile() call will initiate a DMA transfer from the daughter-board receive FIFO, the data-link, to the base-board receive FIFO, to the PCI bus and into host memory.

When the Dart board is recognized by the PCI bus configuration utility it will start the DartBase driver which will create a device object for each board, initialize the hardware; create child devices for the eight I/O channels and request loading of the DartChan driver. The DartChan 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 Dart user manual (also referred to as the hardware manual).

Driver Installation
There are several files provided in each driver package. These files include Dart.inf, DartBase.sys, DartBaseGUID.h, DartChanGUID.h, DDDartBase.h, DartChan.sys, DDDartChan.h, dartdb_base.h, dartdb_chan.h, dartdb_dac.h, DartTest.exe, and DartTest source files.

DartBaseGUID.h and DartChanGUID.h are C header files that define the device interface identifiers for the drivers. DDDartBase.h and DDDartChan.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.
Additional header files: dartdb_base.h, dartdb_chan.h and dartdb_dac.h are provided to define the target register offsets and bit definitions for the supplied daughter-board design and DAC SPI functions.

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

To run DartTest, open a command prompt console window and type `DartTest -d0 -?` to display a list of commands (the DartTest.exe file must be in the directory that the window is referencing). The commands are all of the form `DartTest -dn -im` where `n` and `m` are the device number and DartBase driver ioctl number respectively or `DartTest -cn -im` where `n` and `m` are the channel number (0-7) and DartChan 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 Dart.inf, DartBase.sys and DartChan.sys to a floppy disk, CD, or other accessible location.

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

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

**Windows XP Installation**

Copy Dart.inf, DartBase.sys and DartChan.sys to a floppy disk, CD, or other accessible location.

With the Dart 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 Dart 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 DartBaseGUID.h and DartChanGUID.h.

Below is example code for opening handles for DartBase device `devNum`.

```c
// The maximum length of the device name for a given interface
#define MAX_DEVICE_NAME 256

// Handles to the device objects
HANDLE hDartBase = INVALID_HANDLE_VALUE;
HANDLE hDartChan[DART_BASE_NUM_CHANNELS] = {INVALID_HANDLE_VALUE,
                                          INVALID_HANDLE_VALUE,
                                          INVALID_HANDLE_VALUE,
                                          INVALID_HANDLE_VALUE,
                                          INVALID_HANDLE_VALUE,
                                          INVALID_HANDLE_VALUE,
                                          INVALID_HANDLE_VALUE);

// Dart device number
ULONG devNum

// Dart 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 required to get the symbolic link name
DWORD requiredSize;

// Interface data structures for this device
SP_DEVICE_INTERFACE_DATA interfaceData;

// The base device information structure
DART_BASE_DRIVER_DEVICE_INFO info;

// The channel device information structure
DART_CHAN_DRIVER_DEVICE_INFO cinfo;

// Flag indicating success finding correct device
BOOLEAN found = FALSE;
```
hDeviceInfo = SetupDiGetClassDevs(
    (LPGUID)&GUID_DEVINTERFACE_DART_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_DART_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
hDartBase = CreateFile(deviceName,
    GENERIC_READ | GENERIC_WRITE,
    FILE_SHARE_READ | FILE_SHARE_WRITE,
    NULL,
    OPEN_EXISTING,
    NULL,
    NULL);

if (hDartBase == INVALID_HANDLE_VALUE)
{
    printf("**Error: couldn't open %s, (%d)\n", deviceName,
        GetLastError());
    exit(-1);
}
// Read info
if (!DeviceIoControl(hDartBase,
   IOCTL_DART_BASE_GET_INFO,
   NULL,
   0,
   &info,
   sizeof(info),
   &length,
   NULL))
{
    printf("IOCTL_DART_BASE_GET_INFO failed: %d\n", GetLastError());
    exit(-1);
}

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

SetupDiDestroyDeviceInfoList(hDeviceInfo);

hDeviceInfo = SetupDiGetClassDevs(
   (LPGUID)&GUID_DEVINTERFACE_DART_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 < DART_BASE_NUM_CHANNELS)
   // Find the interface for device
   if (!SetupDiEnumDeviceInterfaces(hDeviceInfo,
      NULL,
      (LPGUID)&GUID_DEVINTERFACE_DART_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
hDartChan[chan] = CreateFile(deviceName, GENERIC_READ | GENERIC_WRITE,
FILE_SHARE_READ | FILE_SHARE_WRITE,
NULL,
OPEN_EXISTING,
NULL,
NULL);

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

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

if(cinfo.InstanceNumber / DART_BASE_NUM_CHANNELS == devNum &&
cinfo.InstanceNumber % DART_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.

```c
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
);
```

The IOCTLs defined for the DartBase driver are described below:

**IOCTL_DART_BASE_GET_INFO**

*Function:* Returns the Base Driver version, Xilinx revision, Switch value, Instance number, Master control-link status and Daughter-board Configuration status, Xilinx revision, Design ID and Target control-link status.

*Input:* None

*Output:* DART_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). When the driver initializes, the daughter-board FPGA configuration status is checked and if the driver determines that the FPGA is configured, it will attempt to establish the master/target control-link between the two boards of the module. If this link can be established, the daughter-board status values will be read in order for them to be reported by this call. See DDDartBase.h for the definition of DART_BASE_DRIVER_DEVICE_INFO.

**IOCTL_DART_BASE_SET_CONFIG**

*Function:* Writes a configuration value to the base control register.

*Input:* Value of base control register (unsigned long integer)

*Output:* None

*Notes:* See DDDartBase.h for the relevant channel control bit definitions. Only the bits in BASE_CNTRL_MASK can be controlled by this call. If the daughter-board FPGA has programmed successfully on power-up, the BASE_CNTRL_DB_LINK_EN and BASE_CNTRL_DB_LINK_STRT bits will already be set by the driver initialization code.
IOCTL_DART_BASE_GET_CONFIG
Function: Reads and returns the base control configuration.
Input: None
Output: Value of the base control register (unsigned long integer)
Notes: Returns the values of the bits in BASE_CNTRL_READ_MASK.

IOCTL_DART_BASE_GET_STATUS
Function: Returns the base status values and clears the latched bits.
Input: None
Output: Value of base status register (unsigned long integer)
Notes: The latched bits in BASE_STAT_LATCH_MASK will be cleared if they are set when the status is read.

IOCTL_DART_BASE_SET_TARGET_REG
Function: Writes a value to one of the daughter-board registers.
Input: DART_BASE_TARGET_REG_WRITE structure
Output: None
Notes: The structure has two fields: RegNumber and WriteData. RegNumber is an unsigned character that contains the daughter-board register number to write to and WriteData is an unsigned long integer that contains the value to write to that register. The header files dartdb_base.h, dartdb_chan.h and dartdb_dac.h contain the register numbers and bit defines for the current daughter-board design and DAC registers.

IOCTL_DART_BASE_GET_TARGET_REG
Function: Reads and returns the value contained in a daughter-board register.
Input: Register number (unsigned character)
Output: Register data (unsigned long integer)
Notes: The value returned is the raw value of the specified register; no bits are masked by this call as the daughter-board register functions are user defined.

IOCTL_DART_BASE_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.
IOCTL_DART_BASE_ENABLE_INTERRUPT

Function: Enables the base 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_DART_BASE_DISABLE_INTERRUPT

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

IOCTL_DART_BASE_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 base master interrupt is enabled. This IOCTL is used for development, to test interrupt processing.

IOCTL_DART_BASE_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 user interrupts.

IOCTL_DART_BASE_LOAD_DB

Function: Loads the daughter-board FPGA from a bit-file or forces a reprogram from the on-board Flash PROM.
Input: DB_LOAD structure
Output: None
Notes: The structure contains the file name of the bit-file to load. The file path is defined in the .inf file as DBDesigns under the local “windows” directory. If the input parameter is not included in the call, the driver will force a reprogram from the daughter-board Flash PROM.
IOCTL_DART_BASE_RESET_DB

*Function:* Causes the BASE_CNTRL_DB_RESET bit to be toggled.
*Input:* None
*Output:* None
*Notes:* After the reset bit is asserted and then released, the driver calls the initialize routine to establish the master/target control-link and then read the daughter-board status (Design ID, Xilinx Rev. etc.).

The IOCTLs defined for the DartChan driver are described below:

IOCTL_DART_CHAN_GET_INFO

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

IOCTL_DART_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 DDDartChan.h for the relevant channel control bit definitions. Only the bits in CHAN_CNTRL_MASK can be controlled by this call. Note that the interrupt enable CHAN_CNTRL_RX_ERR_INTEN comprises five error conditions that are reported separately in the status register (credit, code and disparity errors, link loss and receive FIFO overflow).

IOCTL_DART_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_DART_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. As previously stated the credit, code and disparity errors, link loss and receive FIFO overflow all cause the receive error interrupt to occur. They are all symptoms of a data-link dysfunction.
**IOCTL_DART_CHAN_SET_FIFO_LEVELS**

*Function:* Sets the transmitter almost empty and receiver almost full levels for the channel.

*Input:* DART_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 CHAN_CNTRL_URGNT_IN_EN and/or CHAN_CNTRL_URGNT_OUT_EN control bits are set, these levels are used to determine when to give priority to an input or output DMA channel.

**IOCTL_DART_CHAN_GET_FIFO_LEVELS**

*Function:* Returns the transmitter almost empty and receiver almost full levels for the channel.

*Input:* None

*Output:* DART_CHAN_FIFO_LEVELS structure

*Notes:*  

**IOCTL_DART_CHAN_GET_FIFO_COUNTS**

*Function:* Returns the number of data words in the channel’s transmit and receive FIFOs.

*Input:* None

*Output:* DART_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 1025 32-bit words and the receive count can be a maximum of 1028 32-bit words.

**IOCTL_DART_CHAN_RESET_FIFOS**

*Function:* Resets one or both FIFOs for the referenced channel.

*Input:* DART_FIFO_SEL enumeration type

*Output:* None

*Notes:* Resets the channel’s transmit and/or receive FIFO 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_DART_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 a single-word write access to the transmit FIFO instead of using DMA.
**IOCTL_DART_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 a single-word read access to the receive FIFO instead of using DMA.

---

**IOCTL_DART_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_DART_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_DART_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_DART_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_DART_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**

Dart 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**

Dart 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

All information provided is Copyright Dynamic Engineering.