

DYNAMIC ENGINEERING

150 DuBois St., Suite C, Santa Cruz, CA. 95060

831-457-8891, Fax 831-457-4793

sales@dyneng.com

www.dyneng.com

Est. 1988

AscBase & AscChan

Driver Documentation

Win32 Driver Model

Revision A

Corresponding Hardware: Revision A

10-2008-1201

Corresponding Firmware: Revision B

AscBase & AscChan
WDM drivers for the **PCI-ASCB**
Avionics Standard Comm. Bus Rev.D
Tester PCI board and PMC carrier

Dynamic Engineering
150 DuBois, 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.
Manual Revision A. Revised May 5, 2009

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_ASC_BASE_GET_INFO	12
IOCTL_ASC_BASE_SET_CONFIG	12
IOCTL_ASC_BASE_GET_CONFIG	12
IOCTL_ASC_BASE_GET_STATUS	13
IOCTL_ASC_BASE_LOAD_PLL_DATA	13
IOCTL_ASC_BASE_READ_PLL_DATA	13
IOCTL_ASC_BASE_SET_END_COUNT	13
IOCTL_ASC_BASE_GET_END_COUNT	13
IOCTL_ASC_BASE_SET_INT_COUNT	14
IOCTL_ASC_BASE_GET_INT_COUNT	14
IOCTL_ASC_BASE_READ_COUNT	14
IOCTL_ASC_BASE_SET_PKT_DELAY	14
IOCTL_ASC_BASE_GET_PKT_DELAY	14
IOCTL_ASC_BASE_REGISTER_EVENT	14
IOCTL_ASC_BASE_ENABLE_INTERRUPT	15
IOCTL_ASC_BASE_DISABLE_INTERRUPT	15
IOCTL_ASC_BASE_GET_ISR_STATUS	15
IOCTL_ASC_CHAN_GET_INFO	16
IOCTL_ASC_CHAN_SET_CONFIG	16
IOCTL_ASC_CHAN_GET_CONFIG	16
IOCTL_ASC_CHAN_GET_STATUS	16
IOCTL_ASC_CHAN_SET_MEM_OFFSET	16
IOCTL_ASC_CHAN_GET_MEM_OFFSET	16
IOCTL_ASC_CHAN_WRITE_MEM_DATA	17
IOCTL_ASC_CHAN_READ_MEM_DATA	17
IOCTL_ASC_CHAN_GET_ADDRESS	17
IOCTL_ASC_CHAN_REGISTER_EVENT	17
IOCTL_ASC_CHAN_ENABLE_INTERRUPT	17
IOCTL_ASC_CHAN_DISABLE_INTERRUPT	18
IOCTL_ASC_CHAN_GET_ISR_STATUS	18
IOCTL_ASC_CHAN_INITIALIZE	18
Write.....	19
Read.....	19
Warranty and Repair	20
Service Policy	20
Out of Warranty Repairs	20
For Service Contact:	20



Introduction

The AscBase and AscChan drivers are Win32 driver model (WDM) device drivers for the PCI-ASCB from Dynamic Engineering. The PCI-ASCB board has a Spartan3-1500 Xilinx FPGA to implement the PCI interface, data storage, protocol control and status for 2 ASCB-D channels. Each channel has two 16k-byte dual-port RAM blocks for data transmission and reception. There is also a programmable PLL with one clock output that supplies the frame timing clock reference; the I/O clocks are derived from the on-board 40 MHz oscillator.

When the PCI-ASCB is recognized by the PCI bus configuration utility it will start the AscBase and AscChan drivers to allow communication with the device. 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 IO channel memories.

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-ASCB user manual (also referred to as the hardware manual).

Driver Installation

There are several files provided in each driver package. These files include AscBase.sys, AscChan.sys, AscBusTester.inf, DDAscBase.h, DDAscChan.h, AscBaseGUID.h, AscChanGUID.h, AscTest.exe, and AscTest source files.

DDAscBase.h and DDAscChan.h are C header files that define the Application Program Interface (API) to the drivers. AscBaseGUID.h and AscChanGUID.h are C header files that define the device interface identifier for the AscBase and AscChan drivers. These files are required at compile time by any application that wishes to interface with the driver, but they are not needed for driver installation.

AscTest.exe is a sample Win32 console application that makes calls into the AscBase and AscChan drivers to test each driver call interactively without actually writing any application code. It is not required during the driver installation.

To run AscTest.exe open a command prompt console window and type a command. Type **AscTest -d0 -?** or **AscTest -c0 -?**. This will display a list of commands for the base and channel drivers (the AscTest.exe file must be in the directory that the window is referencing). The commands are all of the form **AscTest -dn -im** where **n** and **m** are the device number and AscBase driver ioctl number respectively or **AscTest -cn -im** where **n** and **m** are the channel number and AscChan driver ioctl number respectively. This application is intended to test the proper functioning of the driver calls and should not be used for normal operation due to overhead processing costs.



Windows 2000 Installation

Copy AscBusTester.inf, AscBase.sys and AscChan.sys to a floppy disk, CD, or other accessible location.

With the PCI-ASCB 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 or location e.g. **Floppy disk drives.**
- Select **Next**
- The wizard should find the AscBusTester.inf file.
- Select **Next**
- Select **Finish** to close the **Found New Hardware Wizard.**

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

Windows XP Installation

Copy AscBusTester.inf, AscBase.sys and AscChan.sys to a floppy disk, CD, or other accessible location.

With the PCI-ASCB 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 ASCB-D channels and reopen the **New Hardware Wizard.** Proceed as above for each channel as necessary.

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 names obtained from the system.

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

Below is example code for opening handles for device devNum.

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

HANDLE hAscChan[ASC_BASE_NUM_CHANNELS] = {INVALID_HANDLE_VALUE,
                                           INVALID_HANDLE_VALUE};

// ASCB-D device number
ULONG devNum
// ASCB-D 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;
PSP_DEVICE_INTERFACE_DETAIL_DATA pDeviceDetail;
// The base device information structure
ASC_BASE_DRIVER_DEVICE_INFO info;
// The channel device information structure
ASC_CHAN_DRIVER_DEVICE_INFO cinfo;
// Flag indicating success finding correct device
BOOLEAN found = FALSE;

hDeviceInfo = SetupDiGetClassDevs(
    (LPGUID)&GUID_DEVINTERFACE_ASC_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_ASC_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
hAscBase = CreateFile(deviceName,
                      GENERIC_READ   | GENERIC_WRITE,
                      FILE_SHARE_READ | FILE_SHARE_WRITE,
                      NULL,
                      OPEN_EXISTING,
                      NULL,
                      NULL);

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

    exit(-1);
}

```

```

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

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

SetupDiDestroyDeviceInfoList(hDeviceInfo);

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

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

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

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

    i++;
}

```

IO Controls

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

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

The IOCTLs defined for the AscBase driver are described below:

IOCTL_ASC_BASE_GET_INFO

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

Input: None

Output: ASC_BASE_DRIVER_DEVICE_INFO structure

Notes: Switch value is the configuration of the on-board dip-switch that has been set by the User (see the board silk screen for bit position and polarity). The PLL ID is the device address of the PLL device. This value, which is set at the factory, is usually 0x69 but may also be 0x6A. See DDAscBase.h for the definition of ASC_BASE_DRIVER_DEVICE_INFO.

IOCTL_ASC_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 DDAscBase.h for the relevant control bit definitions. Only the bits in BASE_CNTRL_MASK can be controlled by this call.

IOCTL_ASC_BASE_GET_CONFIG

Function: 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_ASC_BASE_GET_STATUS

Function: Returns the base interrupt status and clears the latched bits.

Input: None

Output: ASC_BASE_STAT structure

Notes: There are five BOOLEAN fields in the ASC_BASE_STAT structure. They represent the values of the latched status bits in the base status register. The individual interrupt enables in the base control register must be set in order for these to cause an interrupt. Any bits that are returned as TRUE will be cleared by this call. The channel 0 and 1 transmit and receive interrupts are repeated here for convenience. They may alternatively be processed from the individual channels. See DDAscBase.h for the definition of ASC_BASE_STAT

IOCTL_ASC_BASE_LOAD_PLL_DATA

Function: Loads the internal registers of the PLL.

Input: ASC_BASE_PLL_DATA structure

Output: None

Notes: The ASC_BASE_PLL_DATA structure has only one field: Data – an array of 40 bytes containing the values of the PLL device internal registers to write.

IOCTL_ASC_BASE_READ_PLL_DATA

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

Input: None

Output: ASC_BASE_PLL_DATA structure

Notes: The register data is output in the ASC_BASE_PLL_DATA structure in an array of 40 bytes.

IOCTL_ASC_BASE_SET_END_COUNT

Function: Writes a value to the frame end-count register.

Input: Frame-clock counter end-count (unsigned long integer)

Output: None

Notes: An 18-bit counter counts frame clocks. This count determines at what point the counter rolls over to zero again.

IOCTL_ASC_BASE_GET_END_COUNT

Function: Returns the value of the frame end-count register.

Input: None

Output: Frame-clock counter end-count (unsigned long integer)

Notes: Returns the value set in the previous call.

IOCTL_ASC_BASE_SET_INT_COUNT

Function: Writes a value to the frame interrupt count register.

Input: Frame counter interrupt count (unsigned long integer)

Output: None

Notes: This sets the frame-clock counter value that causes the frame-count interrupt status bit to be asserted and can be configured to cause and interrupt if the frame count interrupt enable bit is set in the base control register.

IOCTL_ASC_BASE_GET_INT_COUNT

Function: Returns the value of the frame interrupt count register.

Input: None

Output: Frame counter interrupt count (unsigned long integer)

Notes: Returns the value set in the previous call.

IOCTL_ASC_BASE_READ_COUNT

Function: Returns the current value of the frame-clock counter.

Input: None

Output: Frame-clock count (unsigned long integer)

Notes: Reads and returns the current value of the frame-clock counter.

IOCTL_ASC_BASE_SET_PKT_DELAY

Function: Writes a value to the inter-packet delay register.

Input: Packet delay count (unsigned character)

Output: None

Notes: This 8-bit value is used to determine the inter-packet delay when transmit data packets are sent by the PCI-ASCB. If the register contains zeros, as it will on power-up or after reset, the default value of 0x73 will be used. This results in the nominal inter-packet delay of 9.6 microseconds.

IOCTL_ASC_BASE_GET_PKT_DELAY

Function: Returns the value of the inter-packet delay register.

Input: None

Output: Packet delay count (unsigned character)

Notes: Returns the value set in the previous call.

IOCTL_ASC_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_ASC_BASE_ENABLE_INTERRUPT

Function: Enables the master interrupt.

Input: None

Output: None

Notes: This command must be run to allow the board to respond to local interrupts. The master interrupt enable is disabled in the driver interrupt service routine. Therefore this command must be run after each interrupt occurs to re-enable it.

IOCTL_ASC_BASE_DISABLE_INTERRUPT

Function: Disables the master interrupt.

Input: None

Output: None

Notes: This call is used when local interrupt processing is no longer desired.

IOCTL_ASC_BASE_GET_ISR_STATUS

Function: Returns the interrupt status read in the ISR from the last user interrupt.

Input: None

Output: ASC_BASE_STAT structure

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 latched status bits are cleared in the driver interrupt service routine. See DDAscBase.h for the definition of ASC_BASE_STAT.

The IOCTLs defined for the AscChan driver are described below:

IOCTL_ASC_CHAN_GET_INFO

Function: Returns the driver version and channel instance number.

Input: None

Output: ASC_CHAN_DRIVER_DEVICE_INFO structure

Notes: See DDAscChan.h for the definition of ASC_CHAN_DRIVER_DEVICE_INFO.

IOCTL_ASC_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 DDAscChan.h for the relevant channel control bit definitions. Only the bits in CHAN_CNTRL_MASK can be controlled by this call.

IOCTL_ASC_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_ASC_CHAN_GET_STATUS

Function: Returns the channel's status value and clears the latched bits.

Input: None

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

Notes: The latched bits in CHAN_STAT_LATCH_MASK will be cleared if they are set when the status is read.

IOCTL_ASC_CHAN_SET_MEM_OFFSET

Function: Sets the transmitter or receiver memory offset parameter.

Input: ASC_CHAN_MEMORY_OFFSET structure

Output: None

Notes: Sets the driver parameter used to determine the start of a write or read DMA transfer. See DDAscChan.h for the definition of ASC_CHAN_MEMORY_OFFSET.

IOCTL_ASC_CHAN_GET_MEM_OFFSET

Function: Returns the value of the transmitter or receiver memory offset.

Input: ASC_CHAN_MEM_SEL enumerated type

Output: Memory address offset (unsigned long integer)

Notes: Returns the value last written to the corresponding memory pointer by the previous call. See DDAscChan.h for the definition of ASC_CHAN_MEM_SEL.

IOCTL_ASC_CHAN_WRITE_MEM_DATA

Function: Writes a 32-bit data-word to the transmit or receive memory.

Input: ASC_CHAN_MEMORY_WRITE structure

Output: None

Notes: The structure contains the address to write to and the data to write. The transmit memory address range is 0 – 0x3ffc, the receive memory range is 0x4000 – 0x7ffc. See DDAscChan.h for the definition of ASC_CHAN_MEM_WRITE.

IOCTL_ASC_CHAN_READ_MEM_DATA

Function: Reads and returns a 32-bit data-word from the transmit or receive memory.

Input: Memory address (unsigned long integer)

Output: Memory data (unsigned long integer)

Notes: The transmit memory address range is 0 – 0x3ffc, the receive memory range is 0x4000 – 0x7ffc.

IOCTL_ASC_CHAN_GET_ADDRESS

Function: Returns the next DMA address and current I/O address for the transmitter and receiver.

Input: None

Output: ASC_CHAN_MEMORY_PTR structure

Notes: The structure contains the current addresses that the transmit and receive state-machines are accessing and the next address after the last transmit and receive DMA memory access.

IOCTL_ASC_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_ASC_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_ASC_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_ASC_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.

IOCTL_ASC_CHAN_INITIALIZE

Function: Resets the channel latches and state-machines.

Input: None

Output: None

Notes: This call will stop any I/O or DMA transfers in progress. The dual-port RAMs and channel control register will not be reset, but the transmitter and receiver I/O and DMA enable bits in the channel control register will be cleared.

IOCTL_ASC_CHAN_GET_NEXT_RX_ADDRESS

Function: Returns the first address of the next received packet.

Input: None

Output: Starting address of the next received packet (unsigned short integer)

Notes: This address is updated whenever a receiver packet completes. At the time the address is latched, it is pointing to the address after the last data written for the current packet. This will be where the first status word of the next packet is written.

Write

ASCB-D 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

ASCB-D 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, Suite C
Santa Cruz, CA 95060
831-457-8891 Fax: 831-457-4793
support@dyneng.com

All information provided is Copyright Dynamic Engineering.

