

DYNAMIC ENGINEERING

150 DuBois, Suite 3
Santa Cruz, CA 95060
(831) 457-8891 **Fax** (831) 457-4793
<http://www.dyneng.com>
sales@dyneng.com
Est. 1988

IpCan, BCan & PCan

Driver Documentation

Win32 Driver Model

Revision A
Corresponding Hardware: Revision A/B
10-2006-1101/1102
Corresponding Firmware: Revision A/B

IpCan, BCan & PCan
WDM Device Drivers for the IP-CAN
2-Channel Controller Area Network
Interface IndustryPack® Module

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

©2008 by Dynamic Engineering.
Other trademarks and registered trademarks are owned by their
respective manufactures.
Manual Revision A. Revised March 17, 2008

This document contains information of proprietary interest to Dynamic Engineering. It has been supplied in confidence and the recipient, by accepting this material, agrees that the subject matter will not be copied or reproduced, in whole or in part, nor its contents revealed in any manner or to any person except to meet the purpose for which it was delivered.

Dynamic Engineering has made every effort to ensure that this manual is accurate and complete. Still, the company reserves the right to make improvements or changes in the product described in this document at any time and without notice. Furthermore, Dynamic Engineering assumes no liability arising out of the application or use of the device described herein.

The electronic equipment described herein generates, uses, and can radiate radio frequency energy. Operation of this equipment in a residential area is likely to cause radio interference, in which case the user, at his own expense, will be required to take whatever measures may be required to correct the interference.

Dynamic Engineering's products are not authorized for use as critical components in life support devices or systems without the express written approval of the president of Dynamic Engineering.

Connection of incompatible hardware is likely to cause serious damage.



Table of Contents

Introduction	5
Note	5
Driver Installation.....	6
Windows 2000 Installation	7
Windows XP Installation	7
Driver Startup	8
IO Controls.....	16
IOCTL_IP_CAN_GET_INFO	16
IOCTL_IP_CAN_SET_IP_CONTROL.....	16
IOCTL_IP_CAN_GET_IP_CONTROL	16
IOCTL_IP_CAN_GET_STATUS	17
IOCTL_IP_CAN_RESET_CAN	17
IOCTL_IP_CAN_SET_CAN_MODE	17
IOCTL_IP_CAN_REGISTER_EVENT.....	17
IOCTL_IP_CAN_FORCE_INTERRUPT.....	17
IOCTL_IP_CAN_SET_VECTOR.....	17
IOCTL_IP_CAN_GET_VECTOR.....	18
IOCTL_IP_CAN_ISR_STATUS.....	18
IOCTL_BCAN_GET_INFO	19
IOCTL_BCAN_SET_CONTROL	19
IOCTL_BCAN_GET_CONTROL.....	19
IOCTL_BCAN_GET_STATUS	19
IOCTL_BCAN_RESET_CAN	19
IOCTL_BCAN_GET_CAN_STATUS	20
IOCTL_BCAN_GET_INT_STATUS	20
IOCTL_BCAN_SET_TIMING_CONFIG	20
IOCTL_BCAN_GET_TIMING_CONFIG.....	20
IOCTL_BCAN_SET_ACCEPT_CONFIG.....	20
IOCTL_BCAN_GET_ACCEPT_CONFIG.....	21
IOCTL_BCAN_SET_INTERRUPT_CONFIG.....	21
IOCTL_BCAN_GET_INTERRUPT_CONFIG.....	21
IOCTL_BCAN_SET_COMMAND.....	21
IOCTL_BCAN_REGISTER_EVENT	21
IOCTL_BCAN_ENABLE_INTERRUPT	22
IOCTL_BCAN_DISABLE_INTERRUPT	22
IOCTL_BCAN_FORCE_INTERRUPT.....	22
IOCTL_BCAN_GET_ISR_STATUS	22
Write	23
Read.....	23
IOCTL_PCAN_GET_INFO	24
IOCTL_PCAN_SET_CONTROL	24
IOCTL_PCAN_GET_CONTROL.....	24
IOCTL_PCAN_GET_STATUS	24
IOCTL_PCAN_SET_MODE	25
IOCTL_PCAN_GET_MODE.....	25
IOCTL_PCAN_SET_ERR_COUNT.....	25

IOCTL_PCAN_GET_ERR_COUNT	25
IOCTL_PCAN_GET_CAN_STATUS	25
IOCTL_PCAN_GET_INT_STATUS	26
IOCTL_PCAN_SET_TIMING_CONFIG	26
IOCTL_PCAN_GET_TIMING_CONFIG	26
IOCTL_PCAN_SET_ACCEPT_CONFIG	26
IOCTL_PCAN_GET_ACCEPT_CONFIG	26
IOCTL_PCAN_SET_INTERRUPT_CONFIG	27
IOCTL_PCAN_GET_INTERRUPT_CONFIG	27
IOCTL_PCAN_SET_COMMAND	27
IOCTL_PCAN_REGISTER_EVENT	27
IOCTL_PCAN_ENABLE_INTERRUPT	27
IOCTL_PCAN_DISABLE_INTERRUPT	28
IOCTL_PCAN_FORCE_INTERRUPT	28
IOCTL_PCAN_GET_ISR_STATUS	28
Write	29
Read	29
Warranty and Repair	30
Service Policy	30
Out of Warranty Repairs	30
For Service Contact:	30

Introduction

The IpCan, BCan and PCan drivers are Win32 driver model (WDM) device drivers for the IP-CAN 2-channel Controller Area Network (CAN) Interface IndustryPack® Module from Dynamic Engineering. The IP-CAN board has a Spartan2-50 Xilinx FPGA to implement the Industry Pack interface and protocol control and status for two CAN channels. The CAN devices can operate in one of two modes: BasicCan or PeliCan mode. The BCan driver controls a device in BasicCan mode, while the PCan driver controls a device operating in PeliCan mode. The appropriate driver is loaded automatically for the operating mode selected.

When the IP-CAN is recognized by the system configuration utility it will start the IpCan driver. The IpCan driver enumerates the channels and creates two separate BCan or PCan device objects. This allows the I/O channels to be totally independent while the base driver controls the device items that are common. IO Control calls (IOCTLs) are used to configure the board and read status. Read and Write calls are used to move data in and out of the I/O channel devices. When the CAN devices are first powered-on, or after a hardware reset has been issued, the CAN devices will be in BasicCan mode. If desired an IOCTL call to the IpCan driver can be issued to change the operating mode and the channel driver will be changed appropriately.

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-CAN user manual (also referred to as the hardware manual). Can device data sheet refers to the Phillips SJA1000 Stand-alone CAN controller data sheet. Additional information may be found in application note AN97076 from Phillips.



Driver Installation

Warning: All Dynamic Engineering IndustryPack drivers are only compatible with any of the Dynamic Engineering IP carriers (PCI, CompactPCI or PC104p) and carrier drivers. The appropriate IP carrier driver must be installed before any IP modules can be detected by the system.

There are several files provided in each driver package. These files include IpCan.sys, DDIpCan.h, IpCanGUID.h, BCan.sys, DDBCAn.h, BCanGUID.h, PCan.sys, DDPCAn.h, PCanGUID.h, IpDevice.inf, IpCanTest.exe and IpCanTest source files.

DDIpCan.h, DDBCAn.h and DDPCAn.h are C header files that define the Application Program Interface (API) to the respective drivers. IpCanGUID.h, BCanGUID.h and PCanGUID.h are C header files that define the device interface identifiers for the IpCan, BCan and PCan 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.

IpCanTest.exe is a sample Win32 console application that makes calls into the IpCan and BCan/PCan drivers to test each driver call without actually writing any application code. It also is not required during the driver installation.

To run IpCanTest.exe, open a command prompt console window and type a command. Type ***IpCanTest -d0 -?*** to display a list of commands (the IpCanTest.exe file must be in the directory that the window is referencing). The commands are all of the form ***IpCanTest -dn -im*** where ***n*** and ***m*** are the device number and driver IpCan ioctl number respectively or ***IpCanTest -bn -im*** or ***IpCanTest -pn -im*** where ***n*** and ***m*** are the channel number and BCan/PCan 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.



Windows 2000 Installation

Copy IpDevice.inf, IpCan.sys, BCan.sys and PCan.sys to a floppy disk, or CD if preferred.

With the IpCan hardware installed, power-on the host computer and wait for the **Found New Hardware Wizard** dialogue window to appear.

- Select **Next**.
- Select **Search for a suitable driver for my device**.
- Select **Next**.
- Insert the disk prepared above in the desired drive.
- Select the appropriate drive e.g. **Floppy disk drives**.
- Select **Next**.
- The wizard should find the IpDevice.inf file.
- Select **Next**.
- Select **Finish** to close the **Found New Hardware Wizard**.

The system should now see the IpCan Can channels and reopen the **New Hardware Wizard**. Proceed as above to install the BCan driver for each channel as necessary.

Windows XP Installation

Copy IpDevice.inf, IpCan.sys, BCan.sys and PCan.sys to a floppy disk, or CD if preferred.

With the IpCan hardware installed, power-on the host computer and wait for the **Found New Hardware Wizard** dialogue window to appear.

- Insert the disk prepared above in the desired drive.
- Select **No** when asked to connect to Windows Update.
- Select **Next**.
- Select **Install the software automatically**.
- Select **Next**.
- Select **Finish** to close the **Found New Hardware Wizard**.

The system should now see the IpCan Can channels and reopen the **New Hardware Wizard**. Proceed as above to install the BCan driver for each channel as necessary.

Note: In order to install the PCan driver with either operating system, the operating mode will need to be changed to PeliCan mode for both channels. For convenience, the IpCanTest application can be used for this purpose. The appropriate commands for IpCan board 0 are: **IpCanTest -d0 -i5 0 1** for channel 0 and **IpCanTest -d0 -i5 1 1** for channel 1. If this step is not performed at this time, the system will ask for the driver whenever the operating mode is changed for the first time.

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 globally unique identifiers (GUIDs), which are defined in lpCanGUID.h, BCanGUID.h and PCanGUID.h.

Below is example code for opening handles for device *devNum* (the zero-based device number). To cross-reference the device number to the physical carrier slot in which the lpCan device is installed, use the lpCan GetInfo control call which returns an IP_CAN_DRIVER_DEVICE_INFO structure. The CarrierSwitch field of the structure reports the value of the 8-bit user switch on the IP carrier and the CarrierSlotNum field reports the number of the IP slot on the carrier in which the lpCan is installed (0=slot A, 1=slot B, ...). If more than one carrier is installed in the host computer, the user switches can be set to different values to uniquely identify the target carrier.

Note: In order to build an application with the code below you must link with setupapi.lib and include the following header files: windows.h, stdio.h, stdlib.h, objbase.h, initguid.h, setupapi.h, winerror.h, winioctl.h, process.h and memory.h

```

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

// Handles to device objects
HANDLE hIpCan = INVALID_HANDLE_VALUE;
HANDLE hCan[IP_CAN_NUM_CHANS] = {INVALID_HANDLE_VALUE, INVALID_HANDLE_VALUE};
// IpCan CAN mode - FALSE->BasicCan; TRUE->PeliCan
BOOLEAN pelican = FALSE; // Try BasicCan mode first
// IP-CAN IpCan device number (starting with zero)
ULONG devNum;
// IpCan device instance number (should match devNum, but perhaps not)
ULONG devInst;
// IpCan channel handle array index
ULONG chan;
// Index for interface search loops
UCHAR i;
// Flag to indicate end of channel device search
BOOLEAN done = FALSE;
// Return length from driver call
ULONG length;
// IpCan info structure to match proper instance number
IP_CAN_DRIVER_DEVICE_INFO ipinfo;
// Info structures to match proper channel instance number
BCAN_DRIVER_DEVICE_INFO binfo;
PCAN_DRIVER_DEVICE_INFO pinfo;
// Can channel interface pointer
PVOID pChanInterface;
// Return status from command
LONG status;
// Handle to device interface information structure
HDEVINFO hDeviceInfo;
// The actual symbolic link name to use in the CreateFile() call
CHAR deviceName[MAX_DEVICE_NAME];
// Size of buffer required to get the symbolic link name
DWORD requiredSize;
// Interface data structures for this device
SP_DEVICE_INTERFACE_DATA interfaceData;
PSP_DEVICE_INTERFACE_DETAIL_DATA pDeviceDetail;

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

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

interfaceData.cbSize = sizeof(interfaceData);

```

```

for(i = 0; i <= devNum; i++)
{
    // Find the interface for device devNum
    if(!SetupDiEnumDeviceInterfaces(hDeviceInfo,
                                    NULL,
                                    (LPGUID)&GUID_DEVINTERFACE_IP_CAN,
                                    i,
                                    &interfaceData)

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

// Found our device-get the details data to obtain the symbolic link name
if(!SetupDiGetDeviceInterfaceDetail(hDeviceInfo,
                                    &interfaceData,
                                    NULL,
                                    0,
                                    &requiredSize,
                                    NULL)

{
    if(GetLastError() != ERROR_INSUFFICIENT_BUFFER)
    {
        printf("***Error: couldn't get interface detail, (%d)\n",
                GetLastError());

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

// Allocate a buffer to get detail
pDeviceDetail = (PSP_DEVICE_INTERFACE_DETAIL_DATA)malloc(requiredSize);

if(pDeviceDetail == NULL)
{
    printf("***Error: couldn't allocate interface detail\n");
    SetupDiDestroyDeviceInfoList(hDeviceInfo);
    exit(-1);
}

pDeviceDetail->cbSize = sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);

```

```

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

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

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

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

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

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

if( !DeviceIoControl(hIpCan,
                    IOCTL_IP_CAN_GET_INFO,
                    NULL,
                    0,
                    &ipinfo,
                    sizeof(ipinfo),
                    &length,
                    NULL) )
{
    printf("IOCTL_IP_CAN_GET_INFO failed:  %d\n", GetLastError());
    exit(-1);
}

devInst = ipinfo.InstanceNumber;

interfaceData.cbSize = sizeof(interfaceData);

```

```

i      = 0;
chan  = 0;
done  = FALSE;

while(!done)
{
    // Find the interface for channel devices
    if(pelican)
        pChanInterface = (PVOID)&GUID_DEVINTERFACE_PCAN;
    else
        pChanInterface = (PVOID)&GUID_DEVINTERFACE_BCAN;

    hDeviceInfo = SetupDiGetClassDevs(
        (LPGUID)pChanInterface,
        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);
    }

    if(!SetupDiEnumDeviceInterfaces(hDeviceInfo,
        NULL,
        (LPGUID)pChanInterface,
        i,
        &interfaceData))
    {
        status = GetLastError();
        if(status == ERROR_NO_MORE_ITEMS)
        {
            SetupDiDestroyDeviceInfoList(hDeviceInfo);
            if(!i)
            {
                // No devices of this mode found - try other mode
                pelican = !pelican;
                continue;
            }
            printf("***Error: couldn't find device(no more items), (%d)\n", i);
            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
hCan[chan] = CreateFile(deviceName,
                       GENERIC_READ   | GENERIC_WRITE,
                       FILE_SHARE_READ | FILE_SHARE_WRITE,
                       NULL,
                       OPEN_EXISTING,
                       NULL,
                       NULL);

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

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

// Read info
if(pelican)
{
    if( !DeviceIoControl(hCan[chan],
                        IOCTL_PCAN_GET_INFO,
                        NULL,
                        0,
                        &pinfo,
                        sizeof(pinfo),
                        &length,
                        NULL) )
    {
        printf("IOCTL_PCAN_GET_INFO failed: %d\n", GetLastError());
        SetupDiDestroyDeviceInfoList(hDeviceInfo);
        exit(-1);
    }

    if(pinfo.DeviceNum == (USHORT)devInst)
    {
        if(pinfo.Channel == chan)
        {
            chan++;
            if(IP_CAN_NUM_CHANS == chan)
                done = TRUE;
        }
        else
        {
            printf("\nChannels out of order!\n");
            SetupDiDestroyDeviceInfoList(hDeviceInfo);
            exit(-1);
        }
    }
}
}

```

```

else
{
    if( !DeviceIoControl(hCan[chan],
                        IOCTL_BCAN_GET_INFO,
                        NULL,
                        0,
                        &binfo,
                        sizeof(binfo),
                        &length,
                        NULL) )
    {
        printf("IOCTL_BCAN_GET_INFO failed: %d\n", GetLastError());
        SetupDiDestroyDeviceInfoList(hDeviceInfo);
        exit(-1);
    }

    if(binfo.DeviceNum == (USHORT)devInst)
    {
        if(binfo.Channel == chan)
        {
            chan++;
            if(IP_CAN_NUM_CHANS == chan)
                done = TRUE;
        }
        else
        {
            printf("\nChannels out of order!\n");
            SetupDiDestroyDeviceInfoList(hDeviceInfo);
            exit(-1);
        }
    }
}
i++;
}

// Cleanup
SetupDiDestroyDeviceInfoList(hDeviceInfo);

```

IO Controls

The driver uses IO Control calls (IOCTLs) to configure the device. IOCTLs refer to a single Device Object, which controls a single board 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  
); // used for asynchronous I/O
```

The IOCTLs defined for the IpCan driver are described below:

IOCTL_IP_CAN_GET_INFO

Function: Returns the device instance number, driver version, Xilinx rev., carrier switch value and carrier slot number.

Input: None

Output: IP_CAN_DRIVER_DEVICE_INFO structure

Notes: Instance number is the zero-based module number assigned in the order the devices are enumerated by the system. See DDIpCan.h for the definition of IP_CAN_DRIVER_DEVICE_INFO.

IOCTL_IP_CAN_SET_IP_CONTROL

Function: Sets the configuration of the IP slot.

Input: Register configuration (unsigned long integer)

Output: None

Notes: Controls the IP clock speed for the IP slot that the board occupies. See the bit definitions in DDIpCan.h for more information.

IOCTL_IP_CAN_GET_IP_CONTROL

Function: Returns the configuration of the IP slot.

Input: None

Output: Register configuration (unsigned long integer)

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



IOCTL_IP_CAN_GET_STATUS

Function: Returns the status bits in the IP_CAN_STATUS register.

Input: None

Output: Register configuration (unsigned short integer)

Notes: Returns the interrupt and error status for the two Can devices. See the bit definitions in the DDlpCan.h header file for more information.

IOCTL_IP_CAN_RESET_CAN

Function: Does a hardware reset of one of the Can devices.

Input: Can channel to reset (unsigned char)

Output: None

Notes: The Can device will revert to BasicCan mode after a hardware reset.

IOCTL_IP_CAN_SET_CAN_MODE

Function: Selects the operating mode for a Can device.

Input: Can channel and mode (IP_CAN_CHAN_MODE structure)

Output: None

Notes: All handles referencing the channel device must be closed before issuing this command or the device object will not be removed from the system.

IOCTL_IP_CAN_REGISTER_EVENT

Function: Registers an Event object to be signalled 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_IP_CAN_FORCE_INTERRUPT

Function: Causes an IP interrupt to occur.

Input: None

Output: None

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

IOCTL_IP_CAN_SET_VECTOR

Function: Sets the value of the interrupt vector.

Input: unsigned character

Output: None



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

IOCTL_IP_CAN_GET_VECTOR

Function: Returns the current interrupt vector value.

Input: None

Output: unsigned character

Notes:

IOCTL_IP_CAN_ISR_STATUS

Function: Returns the interrupt status and vector read in the last ISR.

Input: None

Output: IP_CAN_INT_STAT structure

Notes: The status contains the interrupt vector and the contents of the status register read in the last ISR execution. Also, if bit 12 is set in the interrupt status, it indicates that a bus error occurred for this IP slot. See DDIpCan.h for the definition of IP_CAN_INT_STAT.

The IOCTLs defined for the BCan driver are described below:

IOCTL_BCAN_GET_INFO

Function: Returns the channel driver version, Xilinx design revision, the device number and the Can channel number.

Input: None

Output: BCAN_DRIVER_DEVICE_INFO structure

Notes: The device number is passed to the channel devices so that the base device and channel device handles can be coordinated to all apply to the same physical module in the application software. See DDBCAn.h for the definition of BCAN_DRIVER_DEVICE_INFO.

IOCTL_BCAN_SET_CONTROL

Function: Sets the controls for the bus transceiver and bus terminations.

Input: BCAN_CONFIG structure

Output: None

Notes: Controls the transceiver enable and stand-by controls and the termination enable (except rev.A Xilinx which is determined by hardware). See DDBCAn.h for the definition of BCAN_CONFIG.

IOCTL_BCAN_GET_CONTROL

Function: Returns the Can channel control configuration.

Input: None

Output: BCAN_STATE structure

Notes: Returns the device enable, interrupt enable, bus transceiver controls and termination enable state. See DDBCAn.h for the definition of BCAN_STATE.

IOCTL_BCAN_GET_STATUS

Function: Returns the Can device interrupt and transceiver error status.

Input: None

Output: BCAN_STATUS structure

Notes: See DDBCAn.h for the definition of BCAN_STATUS.

IOCTL_BCAN_RESET_CAN

Function: Performs a software reset of the Can device.

Input: Channel to reset (unsigned char)

Output: None

Notes: The operating mode and many of the Can internal registers will be unchanged by this call. See the Can device data sheet for more information on which registers are affected.



IOCTL_BCAN_GET_CAN_STATUS

Function: Returns the state of the Can device internal status register.

Input: None

Output: BCAN_CAN_STATUS structure

Notes: See the Can device data sheet for information on the meaning of the status bits. See DDBCAn.h for the definition of BCAN_CAN_STATUS.

IOCTL_BCAN_GET_INT_STATUS

Function: Returns the contents of the Can interrupt register and associated information.

Input: None

Output: BCAN_INT_STAT structure

Notes: If the receive interrupt is asserted, the first byte of the receive buffer will be read and returned in the RxInfo field. This will specify the length of the pending message. See DDBCAn.h for the definition of BCAN_INT_STAT.

IOCTL_BCAN_SET_TIMING_CONFIG

Function: Sets the Can-bus timing parameters.

Input: BCAN_TIMING_CONFIG structure

Output: None

Notes: This call controls the bit-rate, synchronization jump width, the bit sample point and how many times each bit will be sampled. All the values passed are one less than the effective value. See the Can device data sheet for more information. See DDBCAn.h for the definition of BCAN_TIMING_CONFIG.

IOCTL_BCAN_GET_TIMING_CONFIG

Function: Returns the values set in the previous call.

Input: None

Output: BCAN_TIMING_CONFIG structure

Notes: See the Can device data sheet for more information. See DDBCAn.h for the definition of BCAN_TIMING_CONFIG.

IOCTL_BCAN_SET_ACCEPT_CONFIG

Function: Sets the acceptance filter code and mask.

Input: BCAN_ACCEPT_CONFIG structure

Output: None

Notes: The BasicCan mode only compares the first eight bits of the message identifier to determine acceptance. The mask determines which bits will be checked or ignored. See the Can device data sheet for more information. See DDBCAn.h for the definition of BCAN_ACCEPT_CONFIG.



IOCTL_BCAN_GET_ACCEPT_CONFIG

Function: Returns the values set in the previous call.

Input: None

Output: BCAN_ACCEPT_CONFIG structure

Notes: See the Can device data sheet for more information. See DDBCAn.h for the definition of BCAN_ACCEPT_CONFIG.

IOCTL_BCAN_SET_INTERRUPT_CONFIG

Function: Sets the Can device interrupt enables.

Input: BCAN_INT_CONFIG structure

Output: None

Notes: Determines which conditions in the Can device will cause an interrupt. See the Can device data sheet for interrupt condition descriptions. See DDBCAn.h for the definition of BCAN_INT_CONFIG.

IOCTL_BCAN_GET_INTERRUPT_CONFIG

Function: Returns the values set in the previous call.

Input: None

Output: BCAN_INT_CONFIG structure

Notes: See the Can device data sheet for interrupt condition descriptions. See DDBCAn.h for the definition of BCAN_INT_CONFIG.

IOCTL_BCAN_SET_COMMAND

Function: Issues a command to the Can device.

Input: BCAN_COMMAND_SEL enumerated type

Output: None

Notes: Causes the Can device to initiate a function, such as send a message. See the Can device data sheet for command descriptions. See DDBCAn.h for the definition of BCAN_COMMAND_SEL.

IOCTL_BCAN_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_BCAN_ENABLE_INTERRUPT

Function: Enables the Can channel master interrupt.

Input: None

Output: None

Notes: This command must be run to allow the Can channel to generate interrupts. The master interrupt is disabled in the driver interrupt service routine. Therefore this command must be run after each interrupt is processed to re-enable the interrupts.

IOCTL_BCAN_DISABLE_INTERRUPT

Function: Disables the master interrupt.

Input: None

Output: None

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

IOCTL_BCAN_FORCE_INTERRUPT

Function: Causes a Can channel interrupt to occur.

Input: None

Output: None

Notes: Causes an interrupt to be asserted on the IP bus as if it were caused by the Can device. This IOCTL is used for test and development, to test interrupt processing. The channel force interrupt is not implemented in the rev.A Xilinx design.

IOCTL_BCAN_GET_ISR_STATUS

Function: Returns the interrupt status and associated information from the last ISR.

Input: None

Output: Interrupt status value (BCAN_INT_STAT)

Notes: Returns the status that was read in the interrupt service routine for the last Can channel interrupt serviced. The BCAN_INT and BCAN_ERR bits are shifted down three or five positions depending on the Can channel number to make them consistent for each channel. If the IR_RX bit is set in the Can device interrupt register, the first byte of the receiver buffer will be read and returned. A value of 0xff means no info returned.

Write

BCan data is written to the device using the write command. Writes are executed using the Win32 function WriteFile() (see below) and passing in the handle to the target device, a pointer to a pre-allocated buffer containing the data to be written, an unsigned long integer that represents the number of bytes to be transferred, 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 I/O. The BasicCan transmit buffer is only 10 bytes long, therefore that is the maximum length that can be written with a single write command.

```
BOOL WriteFile(  
    HANDLE         hDevice,           // Handle opened with CreateFile()  
    LPVOID         lpBuffer,         // Pointer to write buffer  
    DWORD         nNumberOfBytesToWrite, // Size of write buffer  
    LPDWORD       lpNumberOfBytesWritten, // Pointer to actual length parameter  
    LPOVERLAPPED lpOverlapped,       // Optional pointer to overlapped  
); // structure used for asynchronous I/O
```

Read

BCan data is read from the device using the read command. Reads are executed using the Win32 function ReadFile() (see below) and passing in the handle to the target device, a pointer to a pre-allocated buffer that will contain the data read, an unsigned long integer that represents the number of bytes to be transferred, 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 I/O. The BasicCan receive buffer is only 10 bytes long, therefore that is the maximum length that can be read with a single read command.

```
BOOL ReadFile(  
    HANDLE         hDevice,           // Handle opened with CreateFile()  
    LPVOID         lpBuffer,         // Pointer to read buffer  
    DWORD         nNumberOfBytesToRead, // Size of read buffer  
    LPDWORD       lpNumberOfBytesRead, // Pointer to actual length parameter  
    LPOVERLAPPED lpOverlapped,       // Optional pointer to overlapped  
); // structure used for asynchronous I/O
```

The IOCTLs defined for the PCan driver are described below:

IOCTL_PCAN_GET_INFO

Function: Returns the channel driver version, Xilinx design revision, the device number and the Can channel number.

Input: None

Output: PCAN_DRIVER_DEVICE_INFO structure

Notes: The device number is passed to the channel devices so that the base device and channel device handles can be coordinated to all apply to the same physical module in the application software. See DDPCan.h for the definition of PCAN_DRIVER_DEVICE_INFO.

IOCTL_PCAN_SET_CONTROL

Function: Sets the controls for the bus transceiver and bus terminations.

Input: PCAN_CONFIG structure

Output: None

Notes: Controls the transceiver enable and stand-by controls and the termination enable (except rev.A Xilinx which is determined by hardware). See DDPCan.h for the definition of PCAN_CONFIG.

IOCTL_PCAN_GET_CONTROL

Function: Returns the Can channel control configuration.

Input: None

Output: PCAN_STATE structure

Notes: Returns the device enable, interrupt enable, bus transceiver controls and termination enable state. See DDPCan.h for the definition of PCAN_STATE.

IOCTL_PCAN_GET_STATUS

Function: Returns the Can device interrupt and transceiver error status.

Input: None

Output: PCAN_STATUS structure

Notes: See DDPCan.h for the definition of PCAN_STATUS.

IOCTL_PCAN_SET_MODE

Function: Sets the configuration of the Can device mode register.

Input: PCAN_MODE structure

Output: None

Notes: Controls various operational mode parameters of the Can device. See the Can device data sheet for information on the mode bits. See DDPCan.h for the definition of PCAN_MODE. Unlike the BasicCan driver the reset bit can be explicitly set and cleared to allow setting up the registers that can only be written in reset mode at the same time. If the device is not in reset mode, the driver will automatically assert and deassert the reset for each appropriate configuration call.

IOCTL_PCAN_GET_MODE

Function: Returns the values set in the previous call.

Input: None

Output: PCAN_MODE structure

Notes: See the Can device data sheet for information on the mode bits. See DDPCan.h for the definition of PCAN_MODE.

IOCTL_PCAN_SET_ERR_COUNT

Function: Writes a value to one of the error counters.

Input: PCAN_COUNT_SET structure

Output: None

Notes: Writes a value to either the Tx error, Rx error or error warning level count. See DDPCan.h for the definition of PCAN_COUNT_SET.

IOCTL_PCAN_GET_ERR_COUNT

Function: Returns the value of one of the error counters.

Input: PCAN_ERR_COUNT_SEL enumerated type

Output: Error count (unsigned char)

Notes: Returns the value of either the Tx error, Rx error or error warning level count. See DDPCan.h for the definition of PCAN_ERR_COUNT_SEL.

IOCTL_PCAN_GET_CAN_STATUS

Function: Returns the state of the Can device internal status register.

Input: None

Output: PCAN_CAN_STATUS structure

Notes: See the Can device data sheet for information on the meaning of the status bits. See DDPCan.h for the definition of PCAN_CAN_STATUS.

IOCTL_PCAN_GET_INT_STATUS

Function: Returns the contents of the Can interrupt register and associated information.

Input: None

Output: PCAN_INT_STAT structure

Notes: If the receive interrupt is asserted, the first byte of the receive buffer will be read and returned in the RxInfo field. This will specify the length of the pending message. See DDPCan.h for the definition of PCAN_INT_STAT.

IOCTL_PCAN_SET_TIMING_CONFIG

Function: Sets the Can-bus timing parameters.

Input: PCAN_TIMING_CONFIG structure

Output: None

Notes: This call controls the bit-rate, synchronization jump width, the bit sample point and how many times each bit will be sampled. All the values passed are one less than the effective value. See the Can device data sheet for more information. See DDPCan.h for the definition of PCAN_TIMING_CONFIG.

IOCTL_PCAN_GET_TIMING_CONFIG

Function: Returns the values set in the previous call.

Input: None

Output: PCAN_TIMING_CONFIG structure

Notes: See the Can device data sheet for more information. See DDPCan.h for the definition of PCAN_TIMING_CONFIG.

IOCTL_PCAN_SET_ACCEPT_CONFIG

Function: Sets the acceptance filter code and mask.

Input: PCAN_ACCEPT_CONFIG structure

Output: None

Notes: The BasicCan mode only compares the first eight bits of the message identifier to determine acceptance. The mask determines which bits will be checked or ignored. See the Can device data sheet for more information. See DDPCan.h for the definition of PCAN_ACCEPT_CONFIG.

IOCTL_PCAN_GET_ACCEPT_CONFIG

Function: Returns the values set in the previous call.

Input: None

Output: PCAN_ACCEPT_CONFIG structure

Notes: See the Can device data sheet for more information. See DDPCan.h for the definition of PCAN_ACCEPT_CONFIG.



IOCTL_PCAN_SET_INTERRUPT_CONFIG

Function: Sets the Can device interrupt enables.

Input: PCAN_INT_CONFIG structure

Output: None

Notes: Determines which conditions in the Can device will cause an interrupt. See the Can device data sheet for interrupt condition descriptions. See DDPCan.h for the definition of PCAN_INT_CONFIG.

IOCTL_PCAN_GET_INTERRUPT_CONFIG

Function: Returns the values set in the previous call.

Input: None

Output: PCAN_INT_CONFIG structure

Notes: See the Can device data sheet for interrupt condition descriptions. See DDPCan.h for the definition of PCAN_INT_CONFIG.

IOCTL_PCAN_SET_COMMAND

Function: Issues a command to the Can device.

Input: PCAN_COMMAND_SEL enumerated type

Output: None

Notes: Causes the Can device to initiate a function, such as send a message. See the Can device data sheet for command descriptions. See DDPCan.h for the definition of PCAN_COMMAND_SEL.

IOCTL_PCAN_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_PCAN_ENABLE_INTERRUPT

Function: Enables the Can channel master interrupt.

Input: None

Output: None

Notes: This command must be run to allow the Can channel to generate interrupts. The master interrupt is disabled in the driver interrupt service routine. Therefore this command must be run after each interrupt is processed to re-enable the interrupts.

IOCTL_PCAN_DISABLE_INTERRUPT

Function: Disables the master interrupt.

Input: None

Output: None

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

IOCTL_PCAN_FORCE_INTERRUPT

Function: Causes a Can channel interrupt to occur.

Input: None

Output: None

Notes: Causes an interrupt to be asserted on the IP bus as if it were caused by the Can device. This IOCTL is used for test and development, to test interrupt processing. The channel force interrupt is not implemented in the rev.A Xilinx design.

IOCTL_PCAN_GET_ISR_STATUS

Function: Returns the interrupt status and associated information from the last ISR.

Input: None

Output: Interrupt status value (PCAN_INT_STAT)

Notes: Returns the status that was read in the interrupt service routine for the last Can channel interrupt serviced. The PCAN_INT and PCAN_ERR bits are shifted down three or five positions depending on the Can channel number to make them consistent for each channel. If the IR_RX bit is set in the Can device interrupt register, the frame information byte of the receiver buffer will be read and returned. If the IR_LARB bit is set in the Can device interrupt register, the Arbitration Lost Capture register will be read and returned. If the IR_BSERR bit is set in the Can device interrupt register, the Error Code Capture will be read and returned. A value of 0xff means no info returned.

Write

PCan data is written to the device using the write command. Writes are executed using the Win32 function WriteFile() (see below) and passing in the handle to the target device, a pointer to a pre-allocated buffer containing the data to be written, an unsigned long integer that represents the number of bytes to be transferred, 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 I/O. The PeliCan transmit buffer is only 13 bytes long, therefore that is the maximum length that can be written with a single write command.

```
BOOL WriteFile(  
    HANDLE         hDevice,           // Handle opened with CreateFile()  
    LPVOID         lpBuffer,         // Pointer to write buffer  
    DWORD         nNumberOfBytesToWrite, // Size of write buffer  
    LPDWORD       lpNumberOfBytesWritten, // Pointer to actual length parameter  
    LPOVERLAPPED  lpOverlapped,      // Optional pointer to overlapped  
); // structure used for asynchronous I/O
```

Read

PCan data is read from the device using the read command. Reads are executed using the Win32 function ReadFile() (see below) and passing in the handle to the target device, a pointer to a pre-allocated buffer that will contain the data read, an unsigned long integer that represents the number of bytes to be transferred, 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 I/O. The PeliCan receive buffer is only 13 bytes long, therefore that is the maximum length that can be read with a single read command.

```
BOOL ReadFile(  
    HANDLE         hDevice,           // Handle opened with CreateFile()  
    LPVOID         lpBuffer,         // Pointer to read buffer  
    DWORD         nNumberOfBytesToRead, // Size of read buffer  
    LPDWORD       lpNumberOfBytesRead, // Pointer to actual length parameter  
    LPOVERLAPPED  lpOverlapped,      // Optional pointer to overlapped  
); // structure used for asynchronous I/O
```

Warranty and Repair

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

Dynamic Engineering's warranty of and liability for defective products is limited to that set forth herein. Dynamic Engineering disclaims and excludes all other product warranties and product liability, expressed or implied, including but not limited to any implied warranties of merchandisability or fitness for a particular purpose or use, liability for negligence in manufacture or shipment of product, liability for injury to persons or property, or for any incidental or consequential damages.

Dynamic Engineering's products are not authorized for use as critical components in life support devices or systems without the express written approval of the president of Dynamic Engineering.

Service Policy

Before returning a product for repair, verify as well as possible that the driver is at fault. The driver has gone through extensive testing and in most cases it will be "cockpit error" rather than an error with the driver. When you are sure or at least willing to pay to have someone help then call the Customer Service Department and arrange to speak with an engineer. We will work with you to determine the cause of the issue. If the issue is one of a defective driver we will correct the problem and provide an updated module(s) to you [no cost]. If the issue is of the customer's making [anything that is not the driver] the engineering time will be invoiced to the customer. Pre-approval may be required in some cases depending on the customer's invoicing policy.

Out of Warranty Repairs

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

For Service Contact:

Customer Service Department
Dynamic Engineering
150 DuBois, Suite 3
Santa Cruz, CA 95060
(831) 457-8891 Fax (831) 457-4793
support@dyneng.com

All information provided is Copyright Dynamic Engineering.

