DYNAMIC ENGINEERING
150 DuBois, Suite C
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

WDF Driver Documentation
For the IP-CAN module

Developed with Windows Driver Foundation Ver1.9

Revision A
Corresponding Hardware: Revision C
10-2006-1103
Corresponding Firmware: Revision C1

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

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

Manual Revision A1. Revised 2/5/19

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

The IpCan, BCan and PCan drivers are Windows device drivers for the IP-CAN 2-channel Controller Area Network (CAN) Interface IndustryPack® Module from Dynamic Engineering. These drivers were developed with the Windows Driver Foundation version 1.9 (WDF) from Microsoft, specifically the Kernel-Mode Driver Framework (KMDF). The IP-CAN board has a Spartan2 Xilinx FPGA to implement the Industry Pack interface and protocol control and status for two CAN channels. The CAN channel devices are implemented with a pair of Phillips SJA-1000's. They can operate in one of two modes: BasicCan or PeliCan mode. The BCan driver controls a device operating in BasicCan mode, while the PCan driver controls a device 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 module drivers are only compatible with one of the Dynamic Engineering IP carriers and carrier drivers (PCI, PCIe, CompactPCI or PC104p). 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, IpCanPublic.h, BCan.sys, BCanPublic.h, PCan.sys, PCanPublic.h, IpPublic.h, IpModDrivers.cat, IpModDrivers.inf and WdfCoInstaller01009.dll.

IpPublic.h, IpCanPublic.h, BCanPublic.h and PCanPublic.h are 'C' header files that define the Application Program Interface (API) to the respective 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.

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

Windows 7 Installation

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

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

- Open the ***Device Manager*** from the control panel.
- Under ***Other devices*** there should be an item for each IP module installed on the IP carrier. The label for a module installed in the first slot of the first PCIe-3IP carrier would read ***PcieCar0 IP Slot A****.
- Right-click on the first device and select ***Update Driver Software***.
- Insert the removable memory device prepared above if necessary.
- Select ***Browse my computer for driver software***.
- Select ***Browse*** and navigate to the memory device or other location prepared above.
- Select ***Next***. The IpCan device driver should now be installed.
- Select ***Close*** to close the update window.

The system should now see the IpCan Can channels. Proceed as above to install the BCan/PCan driver for each channel as necessary. The BCan driver is the default when the IpCan driver is first installed.

Note: To install the PCan driver, a call to IOCTL_IP_CAN_SET_CAN_MODE with the enumerated input parameter set to PELI_CAN must be made.

- Right-click on the remaining IP slot icons and repeat the above procedure as necessary.

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

Driver Startup

Once the driver has been installed it will start automatically when the system recognizes the hardware. A handle can be opened to a specific board by using the CreateFile() function call and passing in the device name obtained from the system. The interface to the devices are identified using globally unique identifiers (GUIDs), which are defined in IpCanPublic.h, BCanPublic.h and PCanPublic.h.

The *main.c* file provided with the user test software can be used as an example to show how to obtain handles to an IpCan device and its I/O channel devices. To cross-reference the device number to the physical carrier slot in which the IpCan device is installed, use the IpCan GetInfo control call which returns a DRIVER_IP_DEVICE_INFO structure. This structure contains information about the IpCan module and the carrier in which it is installed, including a Location string as described in the installation procedure above.

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 information about the IP module, the IP carrier it is installed in and their drivers.
Input: None
Output: IP_CAN_DRIVER_DEVICE_INFO structure
Notes: Instance number is the zero-based module number assigned in the order the IP devices are encountered by the system. If Ip32MCapable is TRUE, then the IP module can operate at either 8 MHz or 32 MHz. If NewIpCntl is TRUE, then the IP module is installed in a PCIe-based carrier and has some additional IP bus control parameters. See the definition of DRIVER_IP_DEVICE_INFO below, which is defined in IpPublic.h.

```
#define IP_LOC_STRING_SIZE    25 // Maximum size of location string (WCHARs)

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

IOCTL_IP_CAN_SET_IP_CONTROL

Function: Sets the IP clock rate and other configuration parameters for the IP slot.
Input: IP_SLOT_CONTROL structure
Output: None
Notes: Some of the fields in this structure are only applicable if the IP-CAN is installed in a PCIe-based carrier slot (NewIpCntl is TRUE). See the definition of IP_SLOT_CONTROL below.

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

IOCTL_IP_CAN_GET_IP_STATE

Function: Returns the configuration parameters for the IP slot that were set by the previous call, as well as several read-only status bits.
Input: None
Output: IP_SLOT_STATE structure
Notes: Returns the slot configuration register value for the IP slot that the board occupies. Interrupt enable and activity status information is also returned. See the definition of IP_SLOT_STATE below.

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

IOCTL_IP_CAN_GET_STATUS

Function: Returns the status bits in the IP_CAN_STATUS register.
Input: None
Output: Status register contents (unsigned short integer)
Notes: Returns the interrupt and error status for the two Can devices. See the bit definitions below for more information.

```
 // IP-Can status register bit defines
#define STAT_LOC_INT        0x0001      // Local interrupt active
#define STAT_CAN_0_INT      0x0010      // CAN 0 interrupt bit
#define STAT_CAN_0_ERR      0x0020      // CAN 0 error bit
#define STAT_CAN_1_INT      0x0040      // CAN 1 interrupt bit
#define STAT_CAN_1_ERR      0x0080      // CAN 1 error bit
```

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 input parameter can only be zero or one. 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.

```
typedef enum _IP_CAN_MODE_SEL {
   BASIC_CAN,
   PELI_CAN
} IP_CAN_MODE_SEL, *PIP_CAN_MODE_SEL;

 // Channel configuration
typedef struct _IP_CAN_CHAN_MODE {
   UCHAR              Channel;
   IP_CAN_MODE_SEL    Mode;
} IP_CAN_CHAN_MODE, *PIP_CAN_CHAN_MODE;
```

IOCTL_IP_CAN_REINIT_CHANS

Function: Causes the CAN channels to be re-enumerated.
Input: None
Output: None
Notes: This call is used to re-evaluate the channel device operating mode after the CAN channel mode has been changed.

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 if the master interrupt is enabled. 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. The vector is 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: Reads the vector storage register and returns the contents.

IOCTL_IP_CAN_ISR_STATUS

Function: Returns the interrupt status and vector read in the last ISR.
Input: None
Output: IP_CAN_ISR_STATUS 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 the definition of IP_CAN_ISR_STATUS below.

```
 // Interrupt status and vector
typedef struct _IP_CAN_ISR_STATUS {
   USHORT   InterruptStatus;
   USHORT   InterruptVector;
} IP_CAN_ISR_STATUS, *PIP_CAN_ISR_STATUS;
```

The IOCTLs defined for the BCan driver are described below:

IOCTL_BCAN_GET_INFO

Function: Returns the channel driver revision, Xilinx design revision, the IpCan 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 below for the definition of BCAN_DRIVER_DEVICE_INFO.

```
 // Driver revision and device instance/channel information
typedef struct _BCAN_DRIVER_DEVICE_INFO {
   UCHAR     DriverRev;
   UCHAR     DeviceNum;
   UCHAR     Channel;
   UCHAR     XilinxRev;
} BCAN_DRIVER_DEVICE_INFO, *PBCAN_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 the definition of BCAN_CONFIG below.

```
typedef struct _BCAN_CONFIG {
   BOOLEAN   TxEnable;
   BOOLEAN   TxStandby;
   BOOLEAN   TermEnable;
} BCAN_CONFIG, *PBCAN_CONFIG;
```

IOCTL_BCAN_GET_STATE

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 the definition of BCAN_STATE below.

```
typedef struct _BCAN_STATE {
   BOOLEAN   TxEnable;
   BOOLEAN   TxStandby;
   BOOLEAN   TermEnable;
   BOOLEAN   CanEnable;
   BOOLEAN   IntEnable;
} BCAN_STATE, *PBCAN_STATE;
```

IOCTL_BCAN_GET_STATUS

Function: Returns the Can device interrupt and transceiver error status.
Input: None
Output: BCAN_STATUS structure
Notes: See the definition of BCAN_STATUS below.

```
typedef struct _BCAN_STATUS {
   BOOLEAN  CanInt;
   BOOLEAN  CanError;
   BOOLEAN  LocalInt;
} BCAN_STATUS, *PBCAN_STATUS;
```

IOCTL_BCAN_RESET_CAN

Function: Performs a software reset of the Can device.
Input: None
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 Can device internal status register values.
Input: None
Output: BCAN_CAN_STATUS structure
Notes: See the Can device data sheet for information on the meaning of the status bits. See the definition of BCAN_CAN_STATUS below.

```
typedef struct _BCAN_CAN_STATUS {
   BOOLEAN  RxAvlb;        // Receive message available
   BOOLEAN  DataOvrn;      // Receive data overrun occurred
   BOOLEAN  TxAvlb;        // Transmit buffer available for write
   BOOLEAN  TxDone;        // Current transmission complete
   BOOLEAN  RxActv;        // Reception in progress
   BOOLEAN  TxActv;        // Transmission in progress
   BOOLEAN  Error;         // An error counter has reached the warning level
   BOOLEAN  BusOff;        // Can not active on bus
} BCAN_CAN_STATUS, *PBCAN_CAN_STATUS;
```

IOCTL_BCAN_GET_INT_STATUS

Function: Returns the contents of the Can interrupt register and associated information.
Input: None
Output: BCAN_INT_STATUS 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. If the receive interrupt is not asserted 0xff will be returned in the RxInfo field. See the definition of BCAN_INT_STATUS below.

```
typedef struct _BCAN_INT_STATUS {
   UCHAR    CanIntReg;
   UCHAR    RxInfo;
} BCAN_INT_STATUS, *PBCAN_INT_STATUS;
```

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 the definition of BCAN_TIMING_CONFIG below.

```
typedef struct _BCAN_TIMING_CONFIG {
   UCHAR    PreScaler;     // 0..63
   UCHAR    SyncJumpWidth; // 0..3
   UCHAR    TimeSeg1;      // 0..15
   UCHAR    TimeSeg2;      // 0..7
   BOOLEAN  Sample3;       // Samples/bit period 1|3
} BCAN_TIMING_CONFIG, *PBCAN_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 the definition of BCAN_TIMING_CONFIG above.

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 the definition of BCAN_ACCEPT_CONFIG below.

```
typedef struct _BCAN_ACCEPT_CONFIG {
   UCHAR     AcceptCode;     // Match against id(10..3)
   UCHAR     AcceptMask;     // b(x)=0->check =1->don't care
} BCAN_ACCEPT_CONFIG, *PBCAN_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 the definition of BCAN_ACCEPT_CONFIG above.

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 the definition of BCAN_INT_CONFIG below.

```
typedef struct _BCAN_INT_CONFIG {
   BOOLEAN  RxIntEn;       // Receive interrupt enable
   BOOLEAN  TxIntEn;       // Transmit interrupt enable
   BOOLEAN  ErrIntEn;      // Error interrupt enable
   BOOLEAN  OvrnIntEn;     // Data overrun interrupt enable
} BCAN_INT_CONFIG, *PBCAN_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 the definition of BCAN_INT_CONFIG above.

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 the definition of BCAN_COMMAND_SEL below.

```
typedef enum _BCAN_COMMAND_SEL {
   BCAN_TREQ,    // Transmission request
   BCAN_TABRT,   // Transmission abort
   BCAN_RRLS,    // Receive buffer release
   BCAN_CLRDO,   // Clear data overrun
   BCAN_SLEEP    // Go-to-sleep request
} BCAN_COMMAND_SEL, *PBCAN_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.

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 Can channel 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 be asserted.
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_ISR_STATUS)
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.

```
#define LOC_INT_ACTV       0x01
#define BCAN_INT           0x02
#define BCAN_ERR           0x04

 // Interrupt register bit defines
#define IR_RX              0x01
#define IR_TX              0x02
#define IR_ERR             0x04
#define IR_OVR             0x08
#define IR_WKUP            0x10

 // Rx buffer length info bit defines
#define RX_DLC_0           0x01
#define RX_DLC_1           0x02
#define RX_DLC_2           0x04
#define RX_DLC_3           0x08
#define RX_RTR             0x10  // Remote transmission request

typedef struct _BCAN_ISR_STATUS {
   UCHAR    IntStatReg;
   UCHAR    CanIntReg;
   UCHAR    RxInfo;
} BCAN_ISR_STATUS, *PBCAN_ISR_STATUS;
```

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 revision, Xilinx design revision, the IpCan 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 for the definition of PCAN_DRIVER_DEVICE_INFO below.

```
// Driver revision and device instance/channel information
typedef struct _PCAN_DRIVER_DEVICE_INFO {
   UCHAR     DriverRev;
   UCHAR     DeviceNum;
   UCHAR     Channel;
   UCHAR     XilinxRev;
} PCAN_DRIVER_DEVICE_INFO, *PPCAN_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 the definition of PCAN_CONFIG below.

```
typedef struct _PCAN_CONFIG {
   BOOLEAN   TxEnable;
   BOOLEAN   TxStandby;
   BOOLEAN   TermEnable;
} PCAN_CONFIG, *PPCAN_CONFIG;
```

IOCTL_PCAN_GET_STATE

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 the definition of PCAN_STATE below.

```
typedef struct _PCAN_STATE {
   BOOLEAN   TxEnable;
   BOOLEAN   TxStandby;
   BOOLEAN   TermEnable;
   BOOLEAN   CanEnable;
   BOOLEAN   IntEnable;
} PCAN_STATE, *PPCAN_STATE;
```

IOCTL_PCAN_GET_STATUS

Function: Returns the Can device interrupt and transceiver error status.
Input: None
Output: PCAN_STATUS structure
Notes: See for the definition of PCAN_STATUS below.

```
typedef struct _PCAN_STATUS {
   BOOLEAN  CanInt;
   BOOLEAN  CanError;
   BOOLEAN  LocalInt;
} PCAN_STATUS, *PPCAN_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 the definition of PCAN_MODE below. 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 in not in reset mode, the driver will automatically assert and deassert the reset for each appropriate configuration call.

```
typedef struct _PCAN_MODE {
   BOOLEAN  ResetRqst;     // Assert software reset
   BOOLEAN  ListenOnly;    // Put Can in listen-only mode
   BOOLEAN  SelfTest;      // Put Can in self-test mode
   BOOLEAN  SingleFilter;  // True=single filter, False=dual filter
   BOOLEAN  GoToSleep;     // Sleep if no int pending or bus activity
} PCAN_MODE, *PPCAN_MODE;
```

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 the definition of PCAN_MODE above.

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 the definitions of PCAN_ERR_COUNT_SEL and PCAN_COUNT_SET below.

```
typedef enum _PCAN_ERR_COUNT_SEL {
   PCAN_TX,
   PCAN_RX,
   PCAN_WARN
} PCAN_ERR_COUNT_SEL, *PPCAN_ERR_COUNT_SEL;

typedef struct _PCAN_COUNT_SET {
   PCAN_ERR_COUNT_SEL   Counter;
   UCHAR                Value;
} PCAN_COUNT_SET, *PPCAN_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 current value of the Tx error, Rx error or error warning level count. See the definition of PCAN_ERR_COUNT_SEL above.

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 the definition of PCAN_CAN_STATUS below.

```
typedef struct _PCAN_CAN_STATUS {
   BOOLEAN  RxAvlb;        // Receive message available
   BOOLEAN  DataOvrn;      // Receive data overrun occurred
   BOOLEAN  TxAvlb;        // Transmit buffer available for write
   BOOLEAN  TxDone;        // Current transmission complete
   BOOLEAN  RxActv;        // Reception in progress
   BOOLEAN  TxActv;        // Transmission in progress
   BOOLEAN  Error;         // An error counter has reached the warning level
   BOOLEAN  BusOff;        // Can not active on bus
} PCAN_CAN_STATUS, *PPCAN_CAN_STATUS;
```

IOCTL_PCAN_GET_INT_STATUS

Function: Returns the contents of the Can interrupt register and associated information.
Input: None
Output: PCAN_INT_STATUS structure
Notes: If the receive interrupt is asserted, the first byte of the receive buffer will be read and returned in the RxFrame field. This will specify the length of the pending message. If the lost arbitration interrupt is asserted, the arbitration lost capture register is read and returned in the AlcCode field. This will specify the bit position where arbitration was lost. If the bus error interrupt is asserted, the error code capture register is read and returned in the EccCode field. This register contains information about the type and location of errors on the bus. See the interrupt bits and PCAN_INT_STATUS definitions below. See the Can device data sheet for more information on these values.

```
#define IR_RX               0x01
#define IR_TX               0x02
#define IR_ERWN             0x04
#define IR_OVR              0x08
#define IR_WKUP             0x10
#define IR_ERPSV            0x20
#define IR_LARB             0x40
#define IR_BSERR            0x80

 typedef struct _PCAN_INT_STATUS {
   UCHAR    CanIntReg;
   UCHAR    RxFrame;
   UCHAR    AlcCode;
   UCHAR    EccCode;
} PCAN_INT_STATUS, *PPCAN_INT_STATUS;
```

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 the definition of PCAN_TIMING_CONFIG below.

```
typedef struct _PCAN_TIMING_CONFIG {
   UCHAR    PreScaler;     // 0..63
   UCHAR    SyncJumpWidth; // 0..3
   UCHAR    TimeSeg1;      // 0..15
   UCHAR    TimeSeg2;      // 0..7
   BOOLEAN  Sample3;       // Samples/bit period 1|3
} PCAN_TIMING_CONFIG, *PPCAN_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 the definition of PCAN_TIMING_CONFIG above.

IOCTL_PCAN_SET_ACCEPT_CONFIG

Function: Sets the acceptance filter code and mask.
Input: PCAN_ACCEPT_CONFIG structure
Output: None
Notes: The PeliCan mode compares up to 32 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 the definition of PCAN_ACCEPT_CONFIG below.

```
typedef struct _PCAN_ACCEPT_CONFIG {
   ULONG    AcceptCode;    // Match against id(28...)
   ULONG    AcceptMask;    // b(x)=0->check =1->don't care
} PCAN_ACCEPT_CONFIG, *PPCAN_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 the definition of PCAN_ACCEPT_CONFIG above.

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 the definition of PCAN_INT_CONFIG below.

```
typedef struct _PCAN_INT_CONFIG {
   BOOLEAN  RxIntEn;       // Receive interrupt enable
   BOOLEAN  TxIntEn;       // Transmit interrupt enable
   BOOLEAN  ErrWrnIntEn;   // Error warning interrupt enable
   BOOLEAN  OvrnIntEn;     // Data overrun interrupt enable
   BOOLEAN  WkIntEn;       // Wake-up interrupt enable
   BOOLEAN  ErrPsvIntEn;   // Error passive interrupt enable
   BOOLEAN  ArbLstIntEn;   // Arbitration lost interrupt enable
   BOOLEAN  BusErrIntEn;   // Bus error interrupt enable
} PCAN_INT_CONFIG, *PPCAN_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 the definition of PCAN_INT_CONFIG above.

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 the definition of PCAN_COMMAND_SEL below.

```
typedef enum _PCAN_COMMAND_SEL {
   PCAN_TREQ,    // Transmission request
   PCAN_TABRT,   // Transmission abort
   PCAN_TSS,     // Single-shot transmission request
   PCAN_RRLS,    // Receive buffer release
   PCAN_CLRDO,   // Clear data overrun
   PCAN_SRREQ,   // Self-reception request
   PCAN_SRSS     // Self-reception single-shot
} PCAN_COMMAND_SEL, *PPCAN_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.

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_ISR_STATUS)
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.

```
#define LOC_INT_ACTV       0x01
#define PCAN_INT           0x02
#define PCAN_ERR           0x04

 // Interrupt register bit defines
#define IR_RX              0x01
#define IR_TX              0x02
#define IR_ERWN            0x04
#define IR_OVR             0x08
#define IR_WKUP            0x10
#define IR_ERPSV           0x20
#define IR_LARB            0x40
#define IR_BSERR           0x80

 // Rx buffer length info bit defines
#define RX_DLC_0           0x01
#define RX_DLC_1           0x02
#define RX_DLC_2           0x04
#define RX_DLC_3           0x08
#define RX_PRTR            0x40  // Remote transmission request
#define RX_FFMT            0x80  // Frame format 1=Extended 0=Standard
```

```
 //Arbitration lost capture register
#define ALC_0              0x01
#define ALC_1              0x02
#define ALC_2              0x04
#define ALC_3              0x08
#define ALC_4              0x10

 // Error code capture register
#define ECC_SEG_0          0x01
#define ECC_SEG_1          0x02
#define ECC_SEG_2          0x04
#define ECC_SEG_3          0x08
#define ECC_SEG_4          0x10
#define ECC_DIR            0x20
#define ECC_TYP_0          0x40
#define ECC_TYP_1          0x80

#define RX_NO_INFO         0xFF
#define ALC_NO_INFO        0xFF
#define ECC_NO_INFO        0xFF

typedef struct _PCAN_ISR_STATUS {
   UCHAR    IntStatReg;
   UCHAR    CanIntReg;
   UCHAR    RxFrame;
   UCHAR    AlcCode;
   UCHAR    EccCode;
} PCAN_ISR_STATUS, *PPCAN_ISR_STATUS;
```

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

http://www.dyneng.com/warranty.html

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. 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