

Intel[®] Platform Controller Hub EG20T

Controller Area Network (CAN) Driver for Windows* Programmer's
Guide

February 2011



INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/#/en_US_01.

Any software source code reprinted in this document is furnished under a software license and may only be used or copied in accordance with the terms of that license.

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. Go to: <http://www.intel.com/products/processor%5Fnumber/> for details.

α Intel® Hyper-Threading Technology requires a computer system with a processor supporting Intel® HT Technology and an Intel® HT Technology-enabled chipset, BIOS and operating system. Performance will vary depending on the specific hardware and software you use. For more information including details on which processors support Intel® HT Technology, see http://www.intel.com/products/ht/hyperthreading_more.htm.

β Intel® High Definition Audio requires a system with an appropriate Intel® chipset and a motherboard with an appropriate CODEC and the necessary drivers installed. System sound quality will vary depending on actual implementation, controller, CODEC, drivers and speakers. For more information about Intel® HD audio, refer to <http://www.intel.com/>.

γ 64-bit computing on Intel® architecture requires a computer system with a processor, chipset, BIOS, operating system, device drivers and applications enabled for Intel® 64 architecture. Performance will vary depending on your hardware and software configurations. Consult with your system vendor for more information.

δ Intel® Virtualization Technology requires a computer system with an enabled Intel® processor, BIOS, virtual machine monitor (VMM) and, for some uses, certain computer system software enabled for it. Functionality, performance or other benefits will vary depending on hardware and software configurations and may require a BIOS update. Software applications may not be compatible with all operating systems. Please check with your application vendor.

ε The original equipment manufacturer must provide Intel® Trusted Platform Module (Intel® TPM) functionality, which requires an Intel® TPM-supported BIOS. Intel® TPM functionality must be initialized and may not be available in all countries.

θ For Enhanced Intel SpeedStep® Technology, see the [Processor Spec Finder](#) or contact your Intel representative for more information.

I²C* is a two-wire communications bus/protocol developed by Philips. SMBus is a subset of the I²C* bus/protocol and was developed by Intel. Implementations of the I²C* bus/protocol may require licenses from various entities, including Philips Electronics N.V. and North American Philips Corporation.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino Inside, Core Inside, i960, Intel, the Intel logo, Intel AppUp, Intel Atom, Intel Atom Inside, Intel Core, Intel Inside, the Intel Inside logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel Sponsors of Tomorrow., the Intel Sponsors of Tomorrow. logo, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, InTru, the InTru logo, InTru soundmark, Itanium, Itanium Inside, MCS, MMX, Moblin, Pentium, Pentium Inside, skool, the skool logo, Sound Mark, The Journey Inside, vPro Inside, VTune, Xeon, and Xeon Inside are trademarks of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2011, Intel Corporation and/or its suppliers and licensors. All rights reserved.



Contents

1.0	Introduction	6
2.0	Operating System (OS) Support	7
3.0	Dependencies	8
4.0	CAN Driver API Details	9
4.1	Features	9
4.2	Interface Details	9
4.3	IOCTL Usage Details	10
4.4	Structures and Enumerations	11
4.4.1	Structures	11
4.4.1.1	ioh_can_msg_t	11
4.4.1.2	ioh_can_timing_t	11
4.4.1.3	ioh_can_error_t	11
4.4.1.4	ioh_can_acc_filter_t	12
4.4.1.5	ioh_can_rx_filter_t	12
4.4.2	Enumerations	12
4.4.2.1	ioh_can_listen_mode_t	12
4.4.2.2	ioh_can_run_mode_t	12
4.4.2.3	ioh_can_arbiter_mode_t	13
4.4.2.4	ioh_can_restart_mode_t	13
4.4.2.5	ioh_can_baud_t	13
4.4.2.6	ioh_can_interrupt_t	13
4.5	Error Handling	14
4.6	Inter-IOCTL dependencies	14
5.0	Programming Guide	15
5.1	Basic Flow	15
5.2	Opening the Device	15
5.2.1	Using GUID Interface Exposed by the Driver	16
5.3	Device Functionality	16
5.3.1	CAN Device Configuration Options	16
5.3.2	CAN Filter Configuration	20
5.3.3	CAN Clock Configuration	21
5.3.4	Getting CAN Error Status	22
5.3.5	Reading and Writing CAN Message	22
5.3.6	Configuring CAN Receive/Transmit Message Object	23
5.3.7	CAN Device Reset	26
5.4	Closing the Device	26

Tables

1	CAN Driver IOCTLs	9
2	ioh_can_msg_t structure	11
3	ioh_can_timing_t structure	11
4	ioh_can_error_t structure	12
5	ioh_can_acc_filter_t structure	12
6	ioh_can_rx_filter_t structure	12
7	ioh_can_listen_mode_t enumeration	12
8	ioh_can_run_mode_t enumeration	13
9	ioh_can_arbiter_mode_t enumeration	13
10	ioh_can_restart_mode_t enumeration	13
11	ioh_can_baud_t enumeration	13



12 ioh_can_interrupt_t enumeration.....14



Revision History

Date	Revision	Description
February 2011	002	Updated Section 2.0 , "Operating System (OS) Support" on page 7 Added Section 5.2.1 , "Using GUID Interface Exposed by the Driver" on page 16
September 2010	001	Initial release

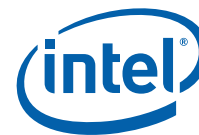


1.0 Introduction

Controller Area Network (CAN) is a multi-master broadcast serial bus standard for connecting electronic control units (ECUs). Each node is able to send and receive messages, but not simultaneously. A message consists primarily of an ID — usually chosen to identify the message-type or sender — and up to eight data bytes. It is transmitted serially onto the bus. This signal pattern is encoded in NRZ and is sensed by all nodes. Devices that are connected by a CAN network are typically sensors, actuators, and other control devices. These devices are not connected directly to the bus, but through a host processor and a CAN controller.

If the bus is free, then any node may begin to transmit. If two or more nodes begin sending messages at the same time, the message with more dominant ID (which has more dominant bits, i.e., zeroes) will overwrite other nodes' less dominant IDs, so that eventually (after this arbitration on the ID) only the dominant message remains and is received by all nodes.

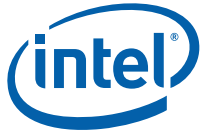
This document describes the CAN driver interfaces exposed to the user mode applications and how to use those interfaces to drive the CAN hardware to achieve effective communication.



2.0 Operating System (OS) Support

The CAN driver is supported by the following operating systems:

No	OS	Notes
1	Microsoft Windows XP*	Service Pack 3
2	Windows Embedded Standard*	2009
3	Windows Embedded POSReady*	2009
4	Microsoft Windows 7*	
5	Windows Embedded Standard7	



3.0 Dependencies

The Intel® Platform Controller Hub EG20T CAN Hardware Assist driver is dependent upon the Intel® Platform Controller Hub EG20T Packet Hub driver to set up the clock frequency before the CAN operation can be started.



4.0 CAN Driver API Details

The CAN driver exposes the interfaces through Input/Output Controls (IOCTLs), which can be called from the user mode applications. The following sections provide information about the IOCTLs of the driver and how to use them to drive the CAN hardware successfully.

4.1 Features

The CAN driver supports:

- 32 message objects
- Setting bit rate up to 1 Mbits/sec
- Enabling the interruption to CAN hardware and setting the interrupt mask and mode
- Disabling the interruption to CAN hardware
- Set or clear selected registers of CAN message object
- Getting the status of selected registers of CAN message object
- Reading the data of selected CAN message object, when the other CAN device responds / event (interrupt) occurs
- Reading CAN hardware and bus status items
- FIFO mode select, programmable FIFO mode (concatenation of message objects) – using FIFO mode or not
- CAN bus byte/multi-byte read transactions
- CAN bus byte/multi-byte write transactions
- Notification that can be sent back to the user mode by using a system wide event object

4.2 Interface Details

Table 1 lists the IOCTLs supported by the CAN driver.

Table 1. CAN Driver IOCTLs (Sheet 1 of 2)

No	Interface	Description
1	IOCTL_CAN_RESET	Reset the CAN device. Issuing this IOCTL causes the device to stop and reset. After the device has been reset, the device must be reconfigured and explicitly set to run.
2	IOCTL_CAN_RUN	Set the device to run. The device must be configured (e.g., timing (baud rate), active/listen mode, etc.) before using this command.
3	IOCTL_CAN_STOP	Stop all operations and the device. The device no longer transmits or receives packets. However, interrupts are still active.
4	IOCTL_CAN_RUN_GET	Get the current run state. This returns whether the device is currently running or stopped.
5	IOCTL_CAN_FILTER	Set the receive filter for a particular receive buffer.
6	IOCTL_CAN_FILTER_GET	Get the receive filter configuration for a particular receive buffer.

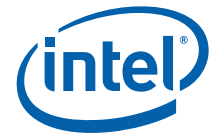


Table 1. CAN Driver IOCTLs (Sheet 2 of 2)

No	Interface	Description
7	IOCTL_CAN_CUSTOM	Set the timing (i.e., baud rate) via custom timing values. Every aspect of the CAN timing must be specified. This should be used when the recommended timings cannot be used.
8	IOCTL_CAN_SIMPLE	Set the timing (baud rate) using one of the pre-defined recommended timings.
9	IOCTL_CAN_TIMING_GET	Get the current timing values.
10	IOCTL_CAN_BLOCK	Set file read and write operation to block. The read operation will wait until a buffer is received before returning. The write operation will not return until the message has been transmitted.
11	IOCTL_CAN_NON_BLOCK	Set file read and write operation to not block. The read operation returns immediately. If a message is waiting, the read operation is returned with that message. Otherwise, the operation returns without any data. The write operation returns immediately, i.e., does not wait until the message is fully transmitted.
12	IOCTL_CAN_BLOCK_GET	Get the blocking state of the device.
13	IOCTL_CAN_LISTEN	Set the device to listen. This allows the device to receive messages, based on the receive buffer filters, but not send any message. This is used for debugging.
14	IOCTL_CAN_ACTIVE	Set the device to active. This is the normal read/write operation of the device.
15	IOCTL_CAN_LISTEN_GET	Get the listen state of the device.
16	IOCTL_CAN_ARBITER_ROUND_ROBIN	Set the transmit buffer arbitration mode to round robin. This means that messages are placed in the next available slot using a round robin scheme (0, 1, ... 7, 0, 1, ...).
17	IOCTL_CAN_ARBITER_FIXED_PRIORITY	Set the transmit buffer arbitration to fixed priority. This means that a message placed in transmit buffer 0 gets the highest priority. Transmit buffer 7 gets the lowest priority.
18	IOCTL_CAN_ARBITER_GET	Get the transmit buffer arbitration mode.
19	IOCTL_CAN_ERROR_STATS_GET	Get the error statistics of the CAN device.
20	IOCTL_CAN_BUFFER_LINK_SET	Set buffer linking for a particular receive buffer.
21	IOCTL_CAN_BUFFER_LINK_CLEAR	Clear the buffer linking for a particular receive buffer.
22	IOCTL_CAN_BUFFER_LINK_GET	Get the receive buffer linking status for a receive buffer.
23	IOCTL_CAN_RX_ENABLE_SET	Enable a receive buffer.
24	IOCTL_CAN_RX_ENABLE_CLEAR	Clear (disable) a receive buffer.
25	IOCTL_CAN_RX_ENABLE_GET	Get the receive buffer enable status.
26	IOCTL_CAN_TX_ENABLE_SET	Enable a transmit buffer.
27	IOCTL_CAN_TX_ENABLE_CLEAR	Clear (disable) a transmit buffer.
28	IOCTL_CAN_TX_ENABLE_GET	Get the transmit buffer enable status.
29	IOCTL_CAN_READ	Read and copy CAN messages to user space.
30	IOCTL_CAN_WRITE	Copy message from user space and transmit.

4.3 IOCTL Usage Details

This section provides the details for configuring the CAN interface and initiating CAN operations. The following files contain the details of the IOCTLs and data structures used for the configuration.



- `ioh_can_ioctl` – contains IOCTL definitions
- `ioh_can_common.h` – data structures and other variables used by the IOCTLs

Refer to [Section 5.0](#) for the programming details.

4.4 Structures and Enumerations

This section provides the structures and enumerations used by interfaces exposed by the CAN driver. All the structures and enumerations used by the interfaces are defined in `ioh_can_common.h`.

4.4.1 Structures

4.4.1.1 `ioh_can_msg_t`

Structure for sending the message data.

Table 2. `ioh_can_msg_t` structure

Name	Description
unsigned short ide	Standard/extended message
unsigned int id	11 or 29 bit msg id
unsigned short dlc	Size of data
Unsigned char data?[IOH_CAN_MSG_DATA_LEN]	Message payload
unsigned short rtr	RTR message

4.4.1.2 `ioh_can_timing_t`

Structure for setting CAN timing values.

Table 3. `ioh_can_timing_t` structure

Name	Description
unsigned int bitrate	Bitrate (kbps)
unsigned int cfg_bitrate	Bitrate value for Baud rate prescaler
unsigned int cfg_tseg1	Timing segment 1
unsigned int cfg_tseg2	Timing segment 2
unsigned int cfg_sjw	Sync jump width
unsigned int smp_l_mode	Sampling mode
unsigned int edge_mode	Edge R / D

4.4.1.3 `ioh_can_error_t`

Structure for getting the CAN error status.



Table 4. **ioh_can_error_t structure**

Name	Description
unsigned int rxgte96	Rx error count >=96
unsigned int txgte96	Tx error count >=96
unsigned int error_stat	Error state of CAN node: 00=error active (normal) 01=error passive 1x=bus off
unsigned int rx_err_cnt	Rx counter
unsigned int tx_err_cnt	Tx counter

4.4.1.4 **ioh_can_acc_filter_t**

Structure contains filter information.

Table 5. **ioh_can_acc_filter_t structure**

Name	Description
unsigned int id	Identifier value
unsigned int id_ext	Standard/extended ID?
unsigned int rtr	RTR message

4.4.1.5 **ioh_can_rx_filter_t**

Structure for setting the CAN message filters.

Table 6. **ioh_can_rx_filter_t structure**

Name	Description
unsigned int num	Message Object Number
unsigned int umask	Mask value
ioh_can_acc_filter_t amr	Mask value for receive message object register
ioh_can_acc_filter_t aidr	Message Identifier value for receive message object

4.4.2 **Enumerations**

This section lists the enumerations exposed by the interface.

4.4.2.1 **ioh_can_listen_mode_t**

CAN listen mode.

Table 7. **ioh_can_listen_mode_t enumeration**

Name	Description
IOH_CAN_ACTIVE	R/w to/from the CAN
IOH_CAN_LISTEN	Only read from the CAN

4.4.2.2 **ioh_can_run_mode_t**

CAN run mode.



Table 8. `ioh_can_run_mode_t` enumeration

Name	Description
IOH_CAN_STOP	CAN stopped
IOH_CAN_RUN	CAN running

4.4.2.3 `ioh_can_arbiter_mode_t`

Identifies valid values for the arbitration mode.

Table 9. `ioh_can_arbiter_mode_t` enumeration

Name	Description
IOH_CAN_ROUND_ROBIN	Equal priority
IOH_CAN_FIXED_PRIORITY	Buffer num priority

4.4.2.4 `ioh_can_restart_mode_t`

Identifies valid values for the auto-restart mode.

Table 10. `ioh_can_restart_mode_t` enumeration

Name	Description
CAN_MANUAL	Manual restart
CAN_AUTO	Automatic restart

4.4.2.5 `ioh_can_baud_t`

Identifies common baudrates.

Table 11. `ioh_can_baud_t` enumeration

Name	Description
IOH_CAN_BAUD_10	10 Kbps
IOH_CAN_BAUD_20	20 Kbps
IOH_CAN_BAUD_50	50 Kbps
IOH_CAN_BAUD_125	125 Kbps
IOH_CAN_BAUD_250	250 Kbps
IOH_CAN_BAUD_500	500 Kbps
IOH_CAN_BAUD_800	800 Kbps
IOH_CAN_BAUD_1000	1000 Kbps

4.4.2.6 `ioh_can_interrupt_t`

Identifies interrupt enable/disable.



Table 12. `ioh_can_interrupt_t` enumeration

Name	Description
CAN_ENABLE	Enable bit only
CAN_DISABLE	Disable bit only
CAN_ALL	All interrupts
CAN_NONE	No interrupt

4.5 Error Handling

Since the IOCTL command is implemented using the Windows* API, the return value of the call is dependent on and defined by the OS. On Windows XP*, the return value is a non-zero value. If the error is detected within or outside the driver, an appropriate system defined value is returned by the driver.

4.6 Inter-IOCTL dependencies

There are no inter-IOCTL dependencies. Once the driver has been loaded successfully, the IOCTLs above can be used in any order.



5.0 Programming Guide

This section explains the basic procedure for using the CAN driver from a user mode application. All operations are through the IOCTLs exposed by the CAN driver. Refer to [Section 4.2](#) for details on the IOCTLs. The steps involved in accessing the CAN driver from the user mode application are described below:

- Opening the Device
- Configure the device for different modes of operations
- Closing the Device

5.1 Basic Flow

The basic flow to use this driver from a user mode application is to open the device, configure the device, transmit/receive messages and then close the device. The following describes the flow using pseudo-code.

```
main()
{
    /* Open the CAN device driver */
    hDevice = CreateFile(DriverName,
                        GENERIC_READ|GENERIC_WRITE,
                        0, NULL, OPEN_EXISTING, 0, NULL);

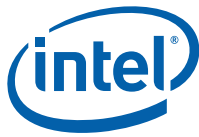
    /* Configure the CAN device driver */
    DeviceIoControl(hDevice, IOCTL_CAN_SIMPLE,
                   &baud, sizeof(baud), NULL, 0, &dwBytesReturned, NULL);

    /* Receive/transmit messages. */
    DeviceIoControl(hDevice, IOCTL_CAN_READ,
                   NULL, 0, &msg, sizeof(msg), &dwBytesReturned, NULL);
    DeviceIoControl(hDevice, IOCTL_CAN_WRITE,
                   &msg, sizeof(msg), NULL, 0, &dwBytesReturned, NULL);

    /* Close the CAN device driver */
    CloseHandle(hDevice);
}
```

5.2 Opening the Device

CAN Device is opened using CreateFile Win32 API. To get the device name, refer to [Section 5.2.1](#).



5.2.1 Using GUID Interface Exposed by the Driver

A device interface class is a way of exporting device and driver functionality to other system components, including other drivers, as well as user-mode applications. A driver can register a device interface class, and then enable an instance of the class for each device object to which user-mode I/O requests might be sent. The Intel® PCH EG20T CAN driver registers the following interface.

No	Interface Name
1	GUID_DEVINTERFACE_IOHCAN

This is defined in `ioh_can_common.h`.

Device interfaces are available to both kernel-mode components and user-mode applications. User-mode code can use **SetupDiXxx** functions to find out the registered, enabled device interfaces.

Please refer the following site to get the details about SetupDiXxx functions.

<http://msdn.microsoft.com/en-us/library/ff549791.aspx>

5.3 Device Functionality

This section describes how to configure the device to achieve time synchronization on Ethernet and CAN.

5.3.1 CAN Device Configuration Options

This section describes the IOCTLs that are used to configure the CAN device and to get different device operation mode statuses.

- **IOCTL_CAN_RESET**
This ioctl is called to reset the device to a known state.

```
bRet = DeviceIoControl( hDevice,  
    IOCTL_CAN_RESET,  
    NULL,  
    0,  
    NULL,  
    0,  
    &dwRet,  
    NULL);
```

- **IOCTL_CAN_RUN**
This ioctl is called to set the device to run mode.

```
bRet = DeviceIoControl( hDevice,  
    IOCTL_CAN_RUN,  
    NULL,  
    0,
```




```

        NULL,
        0,
        &dwRet,
        NULL);
    
```

- **IOCTL_CAN_RUN_GET**

This ioctl is called to get the current run mode of the device.

```

ULONGrunmode;

bRet = DeviceIoControl( hDevice,
        IOCTL_CAN_RUN_GET,
        NULL,
        0,
        &runmode,
        Sizeof(runmode),
        &dwRet,
        NULL);
    
```

- **IOCTL_CAN_BLOCK**

This ioctl is called to put the device to Block mode.

```

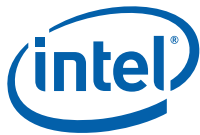
bRet = DeviceIoControl( hDevice,
        IOCTL_CAN_BLOCK,
        NULL,
        0,
        NULL,
        0,
        &dwRet,
        NULL);
    
```

- **IOCTL_CAN_NON_BLOCK**

This ioctl is called to put the device to non-block mode.

```

bRet = DeviceIoControl( hDevice,
        IOCTL_CAN_NON_BLOCK,
        NULL,
        0,
        NULL,
        0,
        &dwRet,
        NULL);
    
```



- **IOCTL_CAN_BLOCK_GET**

This ioctl is called to get the current block mode of the device.

```
ULONG ulBlockGet;  
  
bRet = DeviceIoControl( hDevice,  
                        IOCTL_CAN_BLOCK_GET,  
                        NULL,  
                        0,  
                        &ulBlockGet,  
                        sizeof(ulBlockGet),  
                        &dwRet,  
                        NULL);
```

- **IOCTL_CAN_STOP**

This ioctl is called to put the device to stop mode.

```
bRet = DeviceIoControl( hDevice,  
                        IOCTL_CAN_STOP,  
                        NULL,  
                        0,  
                        NULL,  
                        0,  
                        &dwRet,  
                        NULL);
```

- **IOCTL_CAN_ACTIVE**

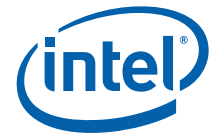
This ioctl is called to put the device to active mode.

```
bRet = DeviceIoControl( hDevice,  
                        IOCTL_CAN_ACTIVE,  
                        NULL,  
                        0,  
                        NULL,  
                        0,  
                        &dwRet,  
                        NULL);
```

- **IOCTL_CAN_LISTEN**

This ioctl is called to put the device to listen mode.

```
bRet = DeviceIoControl( hDevice,  
                        IOCTL_CAN_LISTEN,
```



```

NULL,
0,
NULL,
0,
&dwRet,
NULL);

```

- **IOCTL_CAN_LISTEN_GET**

This ioctl is called to get the current listen mode of the device.

```

ULONG ulListenGet;
bRet = DeviceIoControl( hDevice,
    IOCTL_CAN_LISTEN_GET,
    NULL,
    0,
    &ulListenGet,
    sizeof(ULONG),
    &dwRet,
    NULL);

```

- **IOCTL_CAN_ARBITER_ROUND_ROBIN**

This ioctl is called to put the device priority to round robin.

```

bRet = DeviceIoControl( hDevice,
    IOCTL_CAN_ARBITER_ROUND_ROBIN,
    NULL,
    0,
    NULL,
    0,
    &dwRet,
    NULL);

```

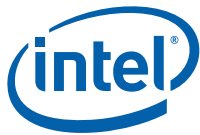
- **IOCTL_CAN_ARBITER_FIXED_PRIORITY**

This ioctl is called to put the device priority to fixed priority.

```

bRet = DeviceIoControl( hDevice,
    IOCTL_CAN_ARBITER_FIXED_PRIORITY,
    NULL,
    0,
    NULL,
    0,

```



```
&dwRet,  
NULL);
```

- **IOCTL_CAN_ARBITER_GET**
This ioctl is called to get the current priority of the device.

```
ULONG ulArbiterGet;  
  
bRet = DeviceIoControl( hDevice,  
    IOCTL_CAN_ARBITER_GET,  
    NULL,  
    0,  
    &ulArbiterGet,  
    sizeof(ULONG),  
    &dwRet,  
    NULL);
```

5.3.2 CAN Filter Configuration

- **IOCTL_CAN_FILTER**
This ioctl is called to set the receive filter for a receive buffer.

```
accFilter.id = 0x01;  
accFilter.id_ext = 0x00;  
accFilter.rtr = 0;  
  
filter.num = 1;  
filter.umask = 0xffff;  
filter.amr = accFilter;  
filter.aidr = accFilter;  
  
bRet = DeviceIoControl( hDevice,  
    IOCTL_CAN_FILTER,  
    &filter,  
    sizeof(ioh_can_rx_filter_t),  
    NULL,  
    0,  
    &dwRet,  
    NULL);
```

- **IOCTL_CAN_FILTER_GET**
This ioctl is called to get the current filter.



```

accFilter.id = 0x01;
accFilter.id_ext = 0x00;
accFilter.rtr=0;

filter.num = 1;
filter.umask = 0xffff;
filter.amr = accFilter;
filter.aidr = accFilter;

bRet = DeviceIoControl( hDevice,
    IOCTL_CAN_FILTER_GET,
    NULL,
    0,
    &filter,
    sizeof(ioh_can_rx_filter_t),
    &dwRet,
    NULL);

```

5.3.3 CAN Clock Configuration

- **IOCTL_CAN_CUSTOM**
This ioctl is called to set the custom clock rate.

```

bRet = DeviceIoControl( hDevice,
    IOCTL_CAN_CUSTOM,
    &NULL,
    0,
    NULL,
    0,
    &dwRet,
    NULL);

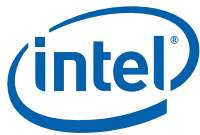
```

- **IOCTL_CAN_SIMPLE**
This ioctl is called to set a different baud rate for the device.

```

ioh_can_baud_t baudrate = IOH_CAN_BAUD_20;
bRet = DeviceIoControl( hDevice,
    IOCTL_CAN_SIMPLE,
    &baudrate,

```



```
    sizeof(baudrate),  
    NULL,  
    0,  
    &dwRet,  
    NULL);
```

- **IOCTL_CAN_TIMING_GET**

This ioctl is called to get the current clock setting of the device.

```
ioh_can_timing_t timing;  
bRet = DeviceIoControl( hDevice,  
    IOCTL_CAN_TIMING_GET,  
    NULL,  
    0,  
    &timing,  
    sizeof(ioh_can_timing_t)  
    &dwRet,  
    NULL);
```

5.3.4 Getting CAN Error Status

- **IOCTL_CAN_ERROR_STATS_GET**

This ioctl is called to get the error status of the device.

```
ioh_can_error_t errorStat;  
bRet = DeviceIoControl( hDevice,  
    IOCTL_CAN_ERROR_STATS_GET,  
    &NULL,  
    0,  
    &errorStat,  
    sizeof(ioh_can_error_t),  
    &dwRet,  
    NULL);
```

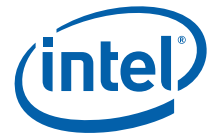
5.3.5 Reading and Writing CAN Message

Following IOCTLs are used for getting receive and transmit CAN messages.

- **IOCTL_CAN_WRITE**

This ioctl is called to write the message.

```
ioh_can_msg_t msg;  
msg.ide=0;
```



```

msg.id=(0x7ff);

msg.dlc=1;

msg.data[0]=10;

msg.rtr=0;

bRet = DeviceIoControl(hDevice,

    IOCTL_CAN_WRITE,

    &msg,

    sizeof(msg),

    NULL,

    0,

    &dwRet,

    NULL);

```

- **IOCTL_CAN_READ**
This ioctl is called to read the message.

```

ioh_can_msg_t msg;

bRet = DeviceIoControl(hDevice,

    IOCTL_CAN_READ,

    NULL,

    0,

    &msg,

    sizeof(msg),

    &dwRet,

    NULL)

```

5.3.6 Configuring CAN Receive/Transmit Message Object

- **IOCTL_CAN_RX_ENABLE_SET**
This ioctl is called to enable the receive buffer.

```

unsigned int uiReceiveBuffNo=1;

bRet = DeviceIoControl( hDevice,

    IOCTL_CAN_RX_ENABLE_SET,

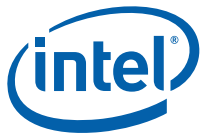
    &uiReceiveBuffNo,

    sizeof(uiReceiveBuffNo),

    NULL,

    0,

```



```
&dwRet,  
NULL);
```

- **IOCTL_CAN_RX_ENABLE_CLEAR**
This ioctl is called to clear the receive buffer.

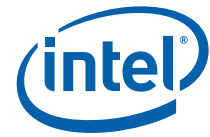
```
unsigned int uiReceiveBuffNo=1;  
bRet = DeviceIoControl( hDevice,  
    IOCTL_CAN_RX_ENABLE_CLEAR,  
    &uiReceiveBuffNo,  
    sizeof(uiReceiveBuffNo),  
    NULL,  
    0,  
    &dwRet,  
    NULL);
```

- **IOCTL_CAN_RX_ENABLE_GET**
This ioctl is called to get the receive buffer status.

```
unsigned int uiReceiveBuffNo=1;  
unsigned int uiStatus;  
bRet = DeviceIoControl( hDevice,  
    IOCTL_CAN_RX_ENABLE_GET,  
    &uiReceiveBuffNo,  
    sizeof(uiReceiveBuffNo),  
    &uiStatus,  
    sizeof(uiStatus),  
    &dwRet,  
    NULL);
```

- **IOCTL_CAN_TX_ENABLE_SET**
This ioctl is called to enable the transmit buffer.

```
unsigned int uiTransmitBuffNo=1;  
bRet = DeviceIoControl( hDevice,  
    IOCTL_CAN_TX_ENABLE_SET,  
    &uiTransmitBuffNo,  
    sizeof(uiTransmitBuffNo),  
    NULL,  
    0,  
    &dwRet,
```

```
NULL);
```

- **IOCTL_CAN_TX_ENABLE_CLEAR**
This ioctl is called to clear the transmit buffer.

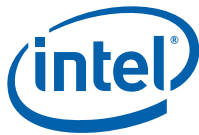
```
unsigned int uiTransmitBuffNo =1;
bRet = DeviceIoControl( hDevice,
    IOCTL_CAN_TX_ENABLE_CLEAR,
    &uiTransmitBuffNo,
    sizeof(uiTransmitBuffNo),
    NULL,
    0,
    &dwRet,
    NULL);
```

- **IOCTL_CAN_TX_ENABLE_GET**
This ioctl is called to get the transmit buffer status.

```
unsigned int uiTransmitBuffNo =1;
unsigned int uiStatus;
bRet = DeviceIoControl( hDevice,
    IOCTL_CAN_RX_ENABLE_GET,
    &uiTransmitBuffNo,
    sizeof(uiTransmitBuffNo),
    &uiStatus,
    sizeof(uiStatus),
    &dwRet,
    NULL);
```

- **IOCTL_CAN_BUFFER_LINK_SET**
This ioctl is called to set the buffer link.

```
unsigned int uiReceiveBuffNo=1;
bRet = DeviceIoControl( hDevice,
    IOCTL_CAN_BUFFER_LINK_SET,
    &uiReceiveBuffNo,
    sizeof(uiReceiveBuffNo),
    NULL,
    0,
    &dwRet,
    NULL);
```



- IOCTL_CAN_BUFFER_LINK_CLEAR
This ioctl is called to clear the buffer link.

```
unsigned int uiReceiveBuffNo=1;

bRet = DeviceIoControl( hDevice,

    IOCTL_CAN_BUFFER_LINK_CLEAR,

    &uiReceiveBuffNo,

    sizeof(uiReceiveBuffNo),

    NULL,

    0,

    &dwRet,

    NULL);
```

- IOCTL_CAN_BUFFER_LINK_GET
This ioctl is called to get the buffer link status.

```
unsigned int uiReceiveBuffNo=1;

unsigned int uiStatus;

bRet = DeviceIoControl( hDevice,

    IOCTL_CAN_BUFFER_LINK_GET,

    &uiReceiveBuffNo,

    sizeof(uiReceiveBuffNo),

    &uiStatus,

    sizeof(uiStatus),

    &dwRet,

    NULL);
```

5.3.7 CAN Device Reset

The following IOCTL is used for resetting the device.

- IOCTL_CAN_RESET

```
bRet = DeviceIoControl( hDevice,

    IOCTL_CAN_RESET, NULL, 0, NULL, 0, &dwRet, NULL);
```

5.4 Closing the Device

Once all the operations related to the CAN driver are finished, the device handle must free the application by calling the Win32 API CloseHandle.

```
CloseHandle(hHandle);
```