Intel® IXDP2400/IXDP2800 Advanced Development Platform

I/O Card Driver API Developer's Manual

January 2004

Document Number: 278671-005

Contents

Figures

Tables

Revision History

Date	Revision	Description
08/30/2002	001	Initial draft from HLD
09/27/2002	002	Review feedback
10/07/2002	003	Update to HLD
10/18/2002	004	Review feedback
10/24/2002	005	API update for Pre-release 5
11/27/2002	006	HLD Update
02/20/2003	007	HLD Update Pre-release 6
06/06/2003	008	LLD Updates - Preliminary Release
July 2003	009	3.1 Beta Release
August 2003	010	IXDP2800 and IXDP2850 beta release
January 2003	011	Added revised IXD2810 Linux driver information.

Document Overview 1

This document describes add-on media card device driver APIs (Application User Interface) for the Intel® IXDP2400 and IXDP2800 advanced development platform. This includes media card driver development for VxWorks and Linux operating systems.

The APIs provide initialization and configuration of the media device including an I/O control path to get various statistics and error counters. The driver does not support any data-path functionality.

In addition to describing device driver functions, this document describes function calling sequences, data structures, components and interfaces.

1.1 Audience

The audience of this guide are software developers who will design, develop, and deliver applications for Intel® IXDP2400 and IXDP2800 advanced development platform. This guide assumes familiarity with the following:

- Realtime network applications
- C Programming

1.2 In This Manual

This manual includes the following chapters:

- Chapter 1 "Document Overview"
  This chapter provides an overview of the document.
- Chapter 2 "Quad Gigabit Ethernet I/O Card"
  This chapter describes the Quad GbE Media Card Driver design for VxWorks and Linux. The driver is implemented as a loadable object module.
- Chapter 3 "10-Port Gigabit Ethernet Media Card"
  This chapter describes the 10-port GbE Media Card Driver design for Linux.
- Chapter 4 "Single OC-192 I/O Card"
  This chapter describes the API module features and API functions.
- Chapter 5 "Single OC-48, Quad OC-12 I/O Card"
  This chapter provides a pointer to information for the Single OC-48, Quad OC-12 I/O Card.

1.3 Other Sources of Information

The Intel® IXP2400 Network Processor and Intel® IXP2800 Network Processor is supported by the following documentation:

- *Intel® IXP2400/IXP2800 Development Tools User's Guide*
- *Help Topics: Developer Workbench*
- *Intel® Internet Exchange Architecture (IXA) Portability Framework Developer's Manual*
- *Intel® Internet Exchange Architecture (IXA) Portability Framework Reference Manual*
  This manual provides details for application development.
- *Intel® Internet Exchange Architecture (IXA) Software Building Blocks Developer's Manual*
- *Intel® Internet Exchange Architecture (IXA) Software Building Blocks Reference Manual*
- *Intel® IXA SDK Release Notes*
- *Intel® IXP2400 Network Processor Datasheet*
- *Intel® IXP2400 Network Processor Hardware Reference Manual*
- *Intel® IXP2400 Network Processor and Intel® IXP2800 Network Processor Getting Started*
- *Intel® IXP2400/IXP2800 Network Processor Programmer's Reference Manual*
- *Intel® Microengine C Compiler Library Reference*

Quad Gigabit Ethernet I/O Card 2

2.1 System Overview

The driver for the Quad Gigabit Ethernet (GbE) I/O card is implemented as a loadable module for both the VxWorks and Linux environments. The GbE I/O card interfaces with the Intel® IXMB2400 Network Processor Base Card which has dual IXP 2400 units.

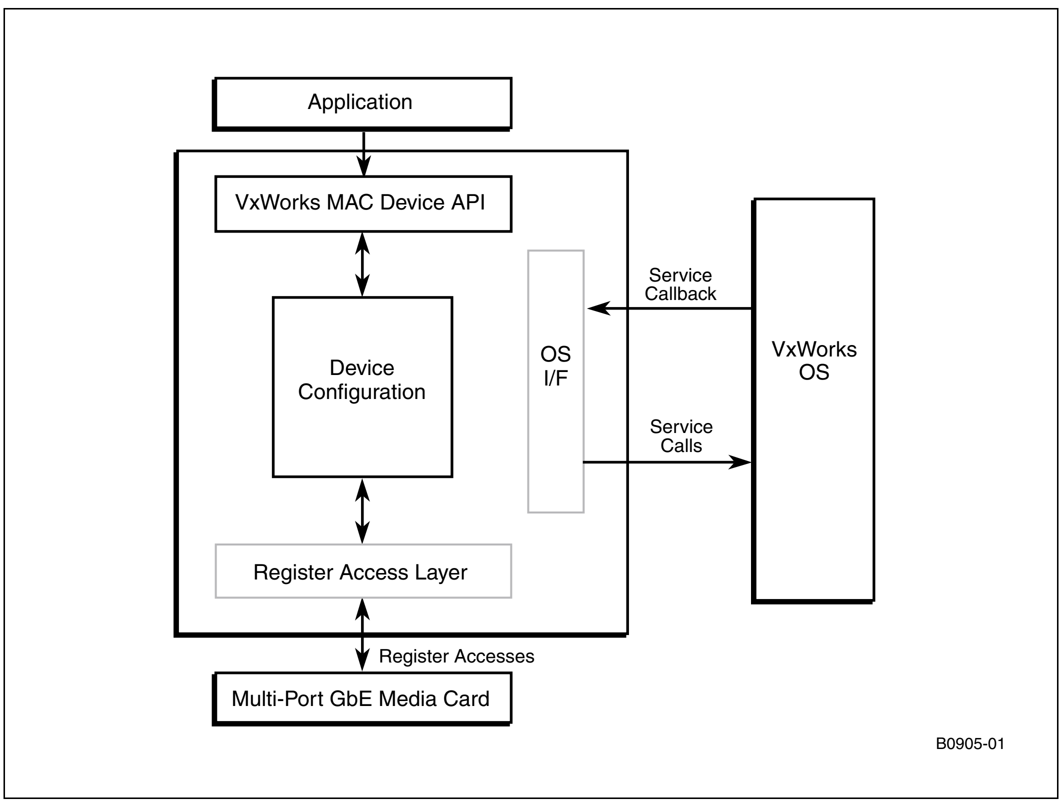
The driver for the Quad GbE I/O Card can run exclusively on the Egress NPU of the Intel® IXMB2400 Network Processor Base Card. Only the Egress NPU has access to slow port of the GbE Media Card.

The main functionality of the driver is initialization and configuration of the Quad GbE I/O card.

2.2 VxWorks Environment

Figure 2-1 shows an overview of the device driver for VxWorks platform, the environment in which the driver is to execute, the major components used in the design, and relationship among the components, followed by component description.

Figure 2-1. VxWorks Driver Architecture



2.2.1 VxWorks MAC Device API

The driver for the Quad GbE I/O Option Card is implemented as a loadable module. The APIs provided by this module are a set of high-level functions that are invoked by applications to initialize and configure the I/O option card.

2.2.2 Device Configuration

The device configuration encompasses initialization and configuration functions. These configuration functions are invoked by the application to configure the Media Card. See Section 2.5, “VxWorks Driver APIs” on page 32 for details.

2.2.3 OS Interface

The driver's OS interface provides functions that let the driver use the OS services related to interrupt handling, memory access etc. This is to ensure the portability and code reuse.

2.2.4 Register access layer

This layer provides functions that read from and write to the device registers. It performs the actual read/write operation on the registers.

2.3 Data Structures

The data structures used for the driver are discussed here. These data structures incorporate the list of `ioctl` commands, used by the user application, to configure the device. The `ioctl` list describes the input and output control commands to configure various registers and get their status. The error enumerator maintains the error number, which is returned by the driver API. The unique error message corresponding to each error number describes the cause and nature of the error.

2.3.1 Basic Data Type

Table 2-1 defines the basic data types that are used in the driver. These types are defined to ease portability of the code across different operating systems.

Table 2-1. Basic Data Type

Basic Types	Description
uint32	32 bit unsigned integer

2.3.2 Structure Passed to ioctl Command

The API provided for calling the `ioctl` command contains a `void` pointer as one of its argument. The calling application passes a structure pointer, which maintains the information regarding the `ioctl` to be called. This structure is passed after typecasting it with the `void *` pointer. This structure, whose pointer is passed by the calling application while using the `ioctl` command, is defined below.

The `ioctl` command normally deals with a single register, that is, 32-bits, and some of the specific `ioctl` commands need to read two, 32-bits registers. Structure definition for a single 32-bit register and two 32-bits registers are described below.

ioctl command structure definition for a single register

```
typedef struct gbe_mac_s_ioctl_ptr {
    uint32 portIndex;
    uint32 value;
} gbe_mac_iotcl_ptr;
```

Input

`portIndex`	Indicates the port number
`value`	A placeholder for a value, which is either to be set or retrieved in an `ioctl` operation.

ioctl command structure definition for two registers

```
typedef struct gbe_mac_s_ioctl_ptr_64 {
    uint32 portIndex;
    uint32 valueHigh;
    uint32 valueLow;
} gbe_mac_iotcl_ptr_64;
```

Note: This structure definition shown for two registers is used in the `GET ioctl` commands such as `GET_STN_ADDR, GET_FDFC_ADDR`, and `GET_MUL_PORT_ADD`, where the 48-bit data for the MAC address is to be read.

2.3.3 IOCTL_CMD Enumerator

The `IOCTL_CMD` enumerator defines `ioctl` code used by the calling application. The application passes the `ioctl` commands as one of the arguments in the `ioctl` function call `GbeMAC_Ioctl()`. The registers of the device are set by using `ioctl` commands prefixed by the word “SET” for setting the register to the given parameter. The `ioctl` commands prefixed by the word “GET” are used to get the register value.

2.3.3.1 MAC Control ioctl Commands

Table 2-2 explains the `ioctl` commands used for configuring and monitoring the status of the registers associated with each MAC port.

Table 2-2. MAC Control ioctl Command (Sheet 1 of 3)

MAC control ioctl commands	Description	Range	Buffer Size
SET_STN_ADDR	Set source MAC address bit 31-0Set source MAC address bit 47-0		64-bits
GET_STN_ADDR	Get source MAC address		64-bits
SET_DUPLEX_MODE	Set half/full-duplex operation mode of the MAC	0x00000000 - 0x00000001	32-bits
GET_DUPLEX_MODE	Get the MAC operating mode	0x00000000 - 0x00000001	32-bits
SET_FDFC_TYPE	Set FDFC Type field of the Transmit Pause Frame	0x00000000 - 0x0000ffff	32-bits
GET_FDFC_TYPE	Get FDFC Type field of the Transmit Pause Frame	0x00000000 - 0x0000ffff	32-bits
SET_COLLISION_DIST	Set limit for the late collisions	0x00000000 - 0x000003ff	32-bits
GET_COLLISION_DIST	Get the limit set for late collision	0x00000000 - 0x000003ff	32-bits
SET_COLLISION_THLD	Set the limit for excessive collision	0x00000000 - 0x000000ff	32-bits
GET_COLLISION_THLD	Get the limit set for excessive collision	0x00000000 - 0x000000ff	32-bits
SET_FCTX_TIMER	Set the pause length sent to the receiving station	0x00000000 - 0x0000ffff	32-bits
GET_FCTX_TIMER	Get the pause length set	0x00000000 - 0x0000ffff	32-bits
SET_FDFC_ADDR	Set 31-0 bits of the 48-bit globally assigned multicast pause frame destination address Set 47-32 bits of the 48-bit globally assigned multicast pause frame destination address		64-bits
GET_FDFC_ADDR	Get the Multicast pause frame destination MAC address		64-bits
SET_IPG_RECEIVE_TIME1	Set the first part of the IPG time for non back-to-back transmission	0x00000000 - 0x000003ff	32-bits
SET_IPG_RECEIVE_TIME2	Set the second part of the IPG time for non back-to-back transmission	0x00000000 - 0x000003ff	32-bits
GET_IPG_RECEIVE_TIME_1	Get the first part of IPG time for non back-to-back transmission	0x00000000 - 0x03ff03ff	32-bits
GET_IPG_RECEIVE_TIME_2	Get the second part of IPG time for non back-to-back transmission	0x00000000 - 0x03ff03ff	32-bits
SET_IPG_TRANSMIT_TIME	Configure IPG time for back-to-back transmission	0x00000000 - 0x000003ff	32-bits
GET_IPG_TRANSMIT_TIME	Get IPG for back-to-back transmission	0x00000000 - 0x000003ff	32-bits

Table 2-2. MAC Control ioctl Command (Sheet 2 of 3)

MAC control ioctl commands	Description	Range	Buffer Size
SET_PAUSE_THRESHOLD	Set the time between two consecutive pause frames to keep the link partner in pause mode.	0x00000000 - 0x0000ffff	32-bits
GET_PAUSE_THRESHOLD	Get the pause threshold time	0x00000000 - 0x0000ffff	32-bits
SET_MAX_FRAME_SIZE	Set the maximum frame size the MAC can receive and transmit without activating any error.	0x00000000 - 0x00003fff	32-bits
GET_MAX_FRAME_SIZE	Get the maximum frame size	0x00000000 - 0x00003fff	32-bits
SET_MAC_IF_MODE	Set the MAC operation frequency and mode per port	0x00000000 - 0x00000007	32-bits
GET_MAC_IF_MODE	Get the MAC operation frequency and mode	0x00000000 - 0x00000007	32-bits
SET_FLUSH_TX	Set this bit to flush all transmit data. It is set if all the traffic to a port should be stopped.	0x00000000 - 0x00000001	32-bits
GET_FLUSH_TX	Get the status of this bit,	0x00000000 - 0x00000001	32-bits
SET_FC_MODE	Set the flow control mode for the RX and TX MAC	0x00000000 - 0x00000007	32-bits
GET_FC_MODE	Get the flow control mode of the RX and TX MAC	0x00000000 - 0x00000007	32-bits
SET_FC_BACK_PRESSURE_LEN	Set the minimum length/duration of backpressure. These six bits holds the value in bytes.	0x00000000 - 0x0000003f	32-bits
GET_FC_BACK_PRESSURE_LEN	Get the minimum length/duration of the backpressure	0x00000000 - 0x0000003f	32-bits
SET_SHORT_RUNT_TH	Set the threshold to determine between short and runt. The 5-bit value holds the value in bytes.	0x00000000 - 0x0000001f	32-bits
GET_SHORT_RUNT_TH	Get the threshold set for the demarcation between short and runt	0x00000000 - 0x0000001f	32-bits
SET_UNKNOWN_FRAME_STT	Used to discard/keep the unknown control frames. Known control frames are pause frames.	0x00000000 - 0x00000001	32-bits
GET_UNKNOWN_FRAME_STT	Check the action regarding the unknown frames	0x00000000 - 0x00000001	32-bits
GET_RX_CONFIG_WORD	This is used in Fiber MAC only for auto negotiation. The contents of this register are the “config_word” received from the link partner	0x00000000 - 0x00bfb1e0	32-bits
SET_TX_CONFIG_WORD	Set this register which is used in Fiber MAC for auto-negotiation only. The contents of this register are sent as the config_word.	0x00000000 - 0x0000ffe0	32-bits
GET_TX_CONFIG_WORD	Get the config_word register contents, sent for auto-negotiation.	0x00000000 - 0x0000ffe0	32-bits
SET_DIV_CONFIG_WORD	Set various configuration bits for general use.	0x00000000 - 0x0001ffff	32-bits

Table 2-2. MAC Control ioctl Command (Sheet 3 of 3)

MAC control ioctl commands	Description	Range	Buffer Size
SET_CHANGE_CONFIG	Change the port configuration value, eg from port configuration Fiber to Copper, or link speed, or mode etc. See Table 2-15, “Mode Value Interpretation” on page 34 for the interpretation of the 0-5 bits for this ioctls.	0x00000000 – 0x0000001F	32-bits
GET_DIV_CONFIG_WORD	Get the various configuration status	0x00000000 - 0x0001ffff	32-bits
SET_PKT_FILTER_CTL	Set this register to allow specific packet types to be marked for filtering. This is used in conjunction with the RX FIFO error Frames Drop Enable Register	0x00000000 - 0x0000003f	32-bits
GET_PKT_FILTER_CTL	Get the status regarding the packet filtering	0x00000000 - 0x0000003f	32-bits
SET_MUL_PORT_ADD	Set bit 31:0 of the address. This address is used to compare against multicast frames at the receiving side if multicast filtering is enabled.Set bit 47:32 of the address		64-bits
GET_MUL_PORT_ADD	Get the Multicast port address.		64-bits
SET_PHY_REGISTER	Set the PHY register value • Bit 0-15 contains the value written to the register • Bit 16-20 represents the PHY register number for the specified port.	0x00000000 – 0x001fffff	32-bits
GET_PHY_REGISTER	Get the PHY register value. In the value, the PHY register number is passed.	0x00000000 – 0x0000ffff	32-bits

2.3.3.2 MAC Receive ioctl Commands

Table 2-3 lists the `ioctl` commands used to monitor the MAC Receive Statistics counters. These `ioctl` Commands can be used in polling and the registers are cleared when read. When RX statistics counter overflows, it gets wrapped back to zero. At the Gbps speed, the 32-bit counters wrap after approximately 30 seconds. The driver polls these registers and accumulates values in virtual 64-bit counters (2-32 bit registers) to ensure that the RX statistics counters do not wrap. For these `ioctl` commands, the calling application must pass 2-32bit registers to get the 64-bit register value.

Table 2-3. MAC Receive Statistics Counters ioctl Commands (Sheet 1 of 2)

MAC RX Stat ioctl Commands	Description	Buffer Size
GET_RX_OCTETS_OK	Get the number of bytes received in all legal frames, including all bytes from the destination MAC address to and including the CRC. The initial preamble and SFD bytes are not counted.	64-bits
GET_RX_OCTETS_BAD	Get the number of bytes received in all bad frames with legal size	64-bits
GET_RX_UC_PKTS	Get the total number of unicast packets received, (EBP)	64-bits
GET_RX_MC_PKTS	Get the total number of multicast packets received (EBP)	64-bits
GET_RX_BC_PKTS	Get the total number of broadcast packets received (EBP)	64-bits

Table 2-3. MAC Receive Statistics Counters ioctl Commands (Sheet 2 of 2)

MAC RX Stat ioctl Commands	Description	Buffer Size
GET_RX_PKTS_64	Get the total number of packets received (IBP) that are 64 octets in length	64-bits
GET_RX_PKTS_65_127	Get the total number of packets received (IBP) that are [65-127] octets in length.	64-bits
GET_RX_PKTS_128_255	Get the total number of packets received (IBP) that are [128-255] octets in length	64-bits
GET_RX_PKTS_256_511	Get the total number of packets received (IBP) that are [256-511] octets in length.	64-bits
GET_RX_PKTS_512_1023	Get the total number of packets received (IBP) that are [512-1023] octets in length	64-bits
GET_RX_PKTS_1024_1518	Get the total number of packets received (IBP) that are [1024-1518] octets in length	64-bits
GET_RX_PKTS_1519_MAX	Get the total number of packets received (IBP) that are >1518 octets in length.	64-bits
GET_RX_FCS_ERR	Get the number of frames, received with legal size, but with wrong CRC field (also called FCS field).	64-bits
GET_VLAN_TAG	Get the number of OK frames with VLAN tag	64-bits
GET_RX_DATA_ERR	Get the number of frames, received with the legal length with code violation.	64-bits
GET_RX_ALLIGN_ERR	Get the number of frames, with a legal frame size, but containing less than 8 additional bits	64-bits
GET_RX_LONG_ERR	Get the number of frames, bigger than the maximum allowed, with both OK CRC and the integral number of octets.	64-bits
GET_RX_JABBER_ERR	Get the number of frames, bigger than the maximum allowed, with either a bad CRC or a non-integral number of octets	64-bits
GET_RX_PAUSE_MAC_CTL	Get the number of Pause MAC control frames received	64-bits
GET_RX_UNKNOWN_CTL_FRAME	Get the number of MAC control frames, received with an op code different from 0001 (Pause)	64-bits
GET_VLONG_ERR	Get the number of frames, bigger than the larger of 2*max frame size and 50000 bits	64-bits
GET_RUNT_ERR	Get the total number of packets, received that are less than 64 octets in length, but longer than or equal to 96 bit times, which corresponds to a 4- byte frame with a well formed preamble and SFD	64-bits
GET_SHORT_ERR	Get the total number of packets, received that are less than 96 bit times, which corresponds to a 4- byte frame with a well formed preamble and SFD.	64-bits
GET_SEQ_ERR	Get the number of sequencing errors that occur in Fiber mode.	64-bits
GET_SYMBOL_ERR	Get the number of symbol errors, encountered by the PHY	64-bits

2.3.3.3 MAC Transmit ioctl Commands

Table 2-4 describes the `ioctl` commands used to monitor the MAC Transmit Statistics counters. These `ioctl` commands can be used in polling. The corresponding registers are cleared when read. When TX statistics counter overflows, it gets wrapped back to zero. At the Gbps speed, the 32-bit counters wrap after approximately 30 seconds. The driver polls these registers and accumulates values in virtual 64-bit counters (two 32-bit registers) to ensure that the RX statistics counters do not wrap. For these `ioctl` commands, the calling application must pass two 32-bit registers to get the 64-bit register value.

Table 2-4. MAC Transmit Statistics Counters ioctl commands (Sheet 1 of 2)

MAC TX Stat ioctl Commands	Description	Buffer Size
`GET_TX_OCTETS_OK`	Get the number of bytes transmitted in all legal frames	64-bits
`GET_TX_OCTETS_BAD`	Get the number of bytes transmitted in all bad frames.	64-bits
`GET_TX_UC_PKTS`	Get the total number of unicast packets transmitted. (EBP)	64-bits
`GET_TX_MC_PKTS`	Get the total number of multicast packets transmitted. (EBP)	64-bits
`GET_TX_BC_PKTS`	Get the total number of broadcast packets transmitted. (EBP)	64-bits
`GET_TX_PKTS_64`	Get the total number of packets transmitted (IBP) that are 64 octets in length	64-bits
`GET_TX_PKTS_65_127`	Get the total number of packets transmitted (IBP) that are [65-127] octets in length	64-bits
`GET_TX_PKTS_128_255`	Get the total number of packets transmitted (IBP) that are [128-255] octets in length	64-bits
`GET_TX_PKTS_256_511`	Get the total number of packets transmitted (IBP) that are [256-511] octets in length	64-bits
`GET_TX_PKTS_512_1023`	Get the total number of packets transmitted (IBP) that are [512 - 1023] octets in length	64-bits
`GET_TX_PKTS_1024_1518`	Get the total number of packets transmitted (IBP) that are [1024-1518] octets in length	64-bits
`GET_TX_PKTS_1519_MAX`	Get the total number of packets transmitted (IBP) that are >1518 octets in length	64-bits
`GET_TX_DEFERRED_ERR`	Get the total number of times, the initial transmission attempt of a frame is postponed due to another frame already being transmitted on the Ethernet network. (HdM)	64-bits
`GET_TX_TOTAL_COLLISION`	Get the sum of all collision events. (HdM)	64-bits
`GET_TX_SINGLE_COLLISION`	Get the number of successfully transmitted frames, on a particular interface where the transmission is inhibited by exactly one collision (HdM)	64-bits
`GET_TX_MUL_COLLISION`	Get the number of successfully transmitted frames, on a particular interface for which transmission is inhibited by more than one collision. (HdM)	64-bits
`GET_LATE_COLLISION`	Get the number of times, a collision is detected on a particular interface later than 512 bit-times into the transmission of a packet. Such frame are terminated and discarded (HdM)	64-bits
`GET_TX_EXCV_COLLISION`	Get the number of frames, which collides 16 times and is then discarded by the MAC. Not effecting Multiple Collisions (HdM)	64-bits

Table 2-4. MAC Transmit Statistics Counters ioctl commands (Sheet 2 of 2)

MAC TX Stat ioctl Commands	Description	Buffer Size
GET_TX_EXCV_DEFERRED_ERR	Get the number of times frame, for which transmission is postponed more than 2*MaxFrameSize due to another frame already being transmitted on the Ethernet network. This causes the MAC to discard the frame. (HdM)	64-bits
GET_TX_EXCV_LEN_DROP	Get the number of frame, for which transmissions aborted by the MAC because the frame is longer than maximum frame size.	64-bits
GET_TX_UNDERRUN	Get the number of internal TX error, which causes the MAC to end the transmission before the end of the frame because the MAC did not get the needed data in time for transmission. The frames are lost and a fragment or a CRC error is transmitted.	64-bits
GET_TX_VLAN_TAG	Get the number of OK frames with VLAN tags.	64-bits
GET_TX_CRC_ERR	Get the number of frames, which are transmitted with a legal size, but with the wrong CRC field (also called FCS field)	64-bits
GET_TX_PAUSE_FRAME	Get the number of Pause frames transmitted.	64-bits
GET_FC_COLLISION_SEND	Get the number of times the collision is generated on purpose on incoming frames, to avoid reception of traffic, while the port is in half-duplex and has flow control enabled, and have not sufficient memory to receive more frames. (HdM)	64-bits

2.3.3.4 Global Status and Configuration ioctl Commands

Table 2-5 lists the `ioctl` commands used for configuration and monitoring the port status.

Table 2-5. Global Status and Configuration Registers ioctl Commands (Sheet 1 of 2)

Global Stat and Config ioctl Commands	Description	Range	Buffer Size
SET_PORT_STATUS	Set the control register for each port in Vallejo device. To make a port active the bit is set to high. Bit 3:0	0x00000000 - 0x0000000f	32-bits
GET_PORT_STATUS	Get the Port status	0x00000000 - 0x0000000f	32-bits
SET_INTERFACE_MODE	Set bit 3:0 1of corresponding register for the PHY interface mode.0 = Fiber, and 1 = Copper	0x00000000 - 0x0000000f	32-bits
GET_INTERFACE_MODE	Get the PHY interface mode for individual port	0x00000000 - 0x0000000f	32-bits
GET_LINK_UP_STATUS	Each bit from 3:0 1of the 32-bit corresponding status register records the status of the Link Flag for a given port. This command reads this to get the status of the individual ports.1 = Link is established	0x00000000 - 0x0000000f	32-bits
GET_RESET_CORE_CLOCK	Get the status of the soft reset for the core cloak system.	0x00000000 - 0x00000001	32-bits
GET_PAUSE_BEHAVIOR	Get the Pause packet behavior	0x00000000 - 0x000f000f	32-bits

Table 2-5. Global Status and Configuration Registers ioctl Commands (Sheet 2 of 2)

Global Stat and Config ioctl Commands	Description	Range	Buffer Size
SET_MAC_SOFT_RESET	Activate per port software reset of the MAC core	0x00000000 - 0x0000000f	32-bits
GET_MAC_SOFT_RESET	Get the status of the software reset of the MAC core.	0x00000000 - 0x0000000f	32-bits
SET_MDIO_RESET	Activate the software reset of the MDIO module	0x00000000 - 0x00000001	32-bits
GET_MDIO_RESET	Get the status regarding the reset activation of the MDIO module	0x00000000 - 0x00000001	32-bits
SET_UI_ENDIAN_MODE	Set microprocessor endian.0 = little endian,1 = big endian	0x00000000 - 0x01000001	32-bits
GET_UI_ENDIAN_MODE	Get microprocessor endian mode	0x00000000 - 0x01000001	32-bits
SET_LED_MODE	Set the LED mode Bit 1: Enable/Disable LED blockBit 0: LED Control	0x00000000 - 0x00000003	32-bits
GET_LED_MODE	Get LED status	0x00000000 - 0x00000003	32-bits
SET_LED_FLASH_RATE	Set LED flash rate,00 = 100 ms flash rate01 = 250 ms flash rate10 = 500 ms flash rate11 = Reserved	0x00000000 - 0x00000003	32-bits
GET_LED_FLASH_RATE	Get LED flash rate	0x00000000 - 0x00000003	32-bits
SET_LED_FAULT_ACTION	Set per-port fault disable/enable the LED flashing for local or remote faults	0x00000000 - 0x0000000f	32-bits
GET_LED_FAULT_ACTION	Get per-port LED fault status	0x00000000 - 0x0000000f	32-bits
GET_JTAG_ID	Get the device identification (fixed here)	0x00450013	32-bits

2.3.3.5 RX FIFO Configuration ioctl Commands

Table 2-6 lists the `ioctl` commands used to configure the status of the receive FIFO.

Table 2-6. ioctl Commands to Configure the RX FIFO (Sheet 1 of 3)

RX FIFO Register ioctl Commands	Description	Range	Buffer Size
SET_RFIFO_HIGH_WATERMARK	Set high watermark for RX FIFO.	0x00000000 - 0x00000fff	32-bits
GET_RFIFO_HIGH_WATERMARK	Get RX FIFO high watermark level.	0x00000000 - 0x00000fff	32-bits
SET_RFIFO_LOW_WATERMARK	Set low watermark for RX FIFO. 2	0x00000000 - 0x00000fff	32-bits
GET_RFIFO_LOW_WATERMARK	Get the RX FIFO low watermark level.	0x00000000 - 0x00000fff	32-bits
GET_RX_FRAME_REMOVED	Get the number of frames lost/removed on individual port when RX FIFO on this port becomes full or reset. 2		32-bits

Table 2-6. ioctl Commands to Configure the RX FIFO (Sheet 2 of 3)

RX FIFO Register ioctl Commands	Description	Range	Buffer Size
`SET_RX_FIFO_PORT_RESET`	Set the soft reset register for each port in the RX block. Bit 3:01	0x00000000 - 0x0000000f	32-bits
`GET_RX_FIFO_PORT_RESET`	Get the soft reset status in the RX block.	0x00000000 - 0x0000000f	32-bits
`SET_RX_FIFO_ERR_FRAME_STT`	Set the action to be taken on receiving error packets, whether such packets are to be dropped or not. Bit 3:011 = Frame Drop Enable0 = Frame Drop Disable	0x00000000 - 0x0000000f	32-bits
`GET_RX_FIFO_ERR_FRAME_STT`	Get the status of the action to be specified on receiving the error packets.	0x00000000 - 0x0000000f	32-bits
`GET_RX_FIFO_OVERFLOW_STT`	Get the RX FIFO status, if a FIFO full situation has occurred. The corresponding register is cleared on read. Bit 3:01	0x00000000 - 0x0000000f	32-bits
`GET_OUT_SEQUENCE_INFO`	Get the status of the RX FIFO, when out of sequence data is detected in the RX FIFO. The corresponding register is cleared on read. Bit 3:01	0x00000000 - 0x0000000f	32-bits
`GET_DROPPED_PKTS`	Get the number of packets dropped by the RX FIFO due to various errors. 2		32-bits
`GET_RW_PTR_RX_FIFO`	Get the value for the read and write pointer for the RX FIFO.2	0x00000000 - 0xffff0fff	32-bits
`GET_OCCUPANCY_RX_FIFO`	Get the occupancy for RX FIFO. The corresponding register is read only. 2	0x00000000 - 0x00001fff	32-bits
`GET_CAPTURED_PKT_LEN`	Get the length information of the captured packet (in bytes) at four ports. The byte position equals to the port number.	0x00000000 - 0xffffffff	32-bits
`SET_INDIRECT_ADR_CTL`	The corresponding register provides the indirect memory access for CPU to read captured data.	0x00000000 - 0x00000fff	32-bits
`GET_INDIRECT_ADR_CTL`	The corresponding register provides the indirect memory access for CPU to read captured data.	0x00000000 - 0x00000fff	32-bits
`GET_READ_DATA`	Get 8 bytes of the read data.		64-bits
`SET_CAPTURE_ENABLE_RX_FIFO`	Set the capture and loop back feature at different ports.Bit 11:81 = Loop back enable.Bit 7:0 = Capture Enable Mode, each pair of bit corresponds to port number from LSB.	0x00000000 - 0x00000fff	32-bits
`GET_CAPTURE_ENABLE_RX_FIFO`	Get the status of the capture enable and loopback feature.	0x00000000 - 0x00000fff	32-bits

Table 2-6. ioctl Commands to Configure the RX FIFO (Sheet 3 of 3)

RX FIFO Register ioctl Commands	Description	Range	Buffer Size
`SET_PRE_PENDING_CRC_ENABLE`	Set the corresponding register to prepend every packet with two extra bytes and also enable the CRC stripping of the packets.Bit 7:41 = Enable CRC stripping.Bit 3:01 = Enable pre-pending, Prepending should not be enabled in loop back mode.	0x00000000 - 0x000000ff	32-bits
`GET_PRE_PENDING_CRC_ENABLE`	Get the status of the pre-pending and CRC stripping feature.	0x00000000 - 0x000000ff	32-bits
`SET_MATCHING_PATTERN`	Set the matching pattern, which is checked with the TYPE/LEN fields of every incoming packet to capture specific packets from data traffic.2	0x00000000 - 0x0000ffff	32-bits
`GET_MATCHING_PATTERN`	Get matching pattern, wet by the previous `ioctl` command.	0x00000000 - 0x0000ffff	32-bits
`SET_JUMBO_PKT_SIZE`	Set the jumbo packet size in 8 byte location. 2	0x00000000 - 0x00000fff	32-bits
`GET_JUMBO_PKT_SIZE`	Get the jumbo packet size set by the previous `ioctl` command.	0x00000000 - 0x00000fff	32-bits
`GET_PKT_DROP_CAP_FIFO`	Get the number of packets dropped at capture FIFO due to FIFO full or bad packets or during CPU not read the previous captured packet. 2		32-bits

2.3.3.6 TX FIFO Configuration ioctl Commands

Table 2-7 lists the `ioctl` commands, used to configure and monitor the transmit FIFO.

Table 2-7. ioctl Commands to Configure and Monitor the TX FIFO (Sheet 1 of 3)

TX FIFO Register ioctl Commands	Description	Range	Buffer Size
`SET_TFIFO_HIGH_WATERMARK`	Set high watermark for TX FIFO, for each port separately.2		32-bits
`GET_TFIFO_HIGH_WATERMARK`	Get high watermark for TX FIFO		32-bits
`SET_TFIFO_LOW_WATERMARK`	Set low watermark for TX FIFO, for each port separately. 2		32-bits
`GET_TFIFO_LOW_WATERMARK`	Get low watermark for TX FIFO.		32-bits
`SET_MAC_THRESHOLD`	Set the MAC threshold for TX FIFO. 2		32-bits
`GET_MAC_THRESHOLD`	Get the MAC threshold TX FIFO value.2		32-bits
`GET_TX_FIFO_OVERFLOW_STT`	Get the status information as Bit 11:81 FIFO out of sequence event trace recordBit 7:41 FIFO underflow event trace recordBit 3:01 FIFO Overflow event trace record.	0x00000000 - 0x00000fff	32-bits

Table 2-7. ioctl Commands to Configure and Monitor the TX FIFO (Sheet 2 of 3)

TX FIFO Register ioctl Commands	Description	Range	Buffer Size
`SET_LOOP_RX_TX`	Set the respective bit high to perform the external loop back.Bit 3:01 0 = Normal Operation1 = The SPI-3 data coming from the RX block is sent to the TX FIFO instead of the SPI-3 Receive interface	0x00000000 - 0x0000000f	32-bits
`GET_LOOP_RX_TX_STT`	Get external loop back status	0x00000000 - 0x0000000f	32-bits
`SET_TX_FIFO_PORT_RESET`	Assert/De-assert reset for each port in TX block.Bit 3:01 set to low to make port active.	0x00000000 - 0x0000000f	32-bits
`GET_TX_TFIFO_PORT_RESET`	Get status of the port	0x00000000 - 0x0000000f	32-bits
`GET_TX_DROP_FRAME`	Get the number of frames lost/removed, when TX FIFO on individual port2 becomes full or reset. This register is clear on read.		32-bits
`GET_TX_DROP_PKTS`	Get the number of packets dropped by the TX FIFO of individual port2, due to various errors. This register is cleared on Read.		32-bits
`GET_TX_RW_PTR`	Get the value of the read write pointer for the TX FIFO of individual port2. This register is cleared on read.	0x00000000 - 0x003fffff	32-bits
`GET_TX_OCCUPANCY`	Get the occupancy for the TX FIFO 2. The corresponding register is read only.		32-bits
`SET_TX_INSERT_DATA`	Insert the 8 bytes data for port 0		64-bits
`GET_TX_INSERT_DATA`	Get the inserted 8-bytes data for each port2 separately.		64-bits
`SET_TX_FIFO_INFO_ADR`	Set the indirect memory access for CPU to write/read data to/from individual insertion FIFO port2.Bit 10 = ResetBit 9 = WriteBit 8 = ReadBit 7:3 = AddressBit 2:0 = Info	0x00000000 - 0x000007ff	32-bits
`GET_TX_FIFO_INFO_ADR`	Get the above defined status2	0x00000000 - 0x000007ff	32-bits
`SET_TX_FIFO_DROP_INSERT`	Enable independently, the individual TX FIFO to drop the erroneous packet and insertion of packet through insertion FIFO.Bit 7:41 = Set high to enable read from insertion FIFO.Bit 3:01 = Set high to discard the error packets in TX FIFO.	0x00000000 - 0x000000ff	32-bits

Table 2-7. ioctl Commands to Configure and Monitor the TX FIFO (Sheet 3 of 3)

TX FIFO Register ioctl Commands	Description	Range	Buffer Size
`GET_TX_FIFO_DROP_INSERT`	Get the above defined feature in corresponding `SET ioctl`.	0c00000000 - 0x000000ff	32-bits
`SET_TX_MINI_FRAME_SIZE`	Set the different minimum length of the packets to be transmitted to MAC independently. These values are used to pad short packets if padding is enabled.Bit 19:161 = Set bit high to enable padding of short packets.Bit 15:12 = (for port 3) If the programmed value is 'N' then the minimum number of bytes in packet is equal to 'N * 8' bytes. Where N = A, B, C, D and ESame as above, bit 11:8,7:4 and 3:0 are for port 2,1,0 respectively.	0x00000000 - 0x000fffff	32-bits
`GET_TX_MINI_FRAME_SIZE`	Get the minimum length of the packet to be transmitted to the MAC.	0x00000000 - 0x000fffff	32-bits

2.3.3.7 MDIO Interface Related ioctl Commands

Table 2-8 lists the `ioctl` commands to configure and monitor the MDIO interface.

Table 2-8. ioctl Commands to Configure and Monitor MDIO Interface

MDIO Interface ioctl Commands	Description	Range	Buffer Size
`SET_MDIO_CMD_ADDR`	Bit 20 = Set high to perform operationBit 17:16 = Identify operation to be performed.Bit 9:8 = address of external deviceBit 4:0 = Reg Address	0x00000000 - 0x0013031f	32-bits
`GET_MDIO_CMD_ADDR`	Get that value of the MDIO command and address register.	0x00000000 - 0x0013031f	32-bits
`SET_MDIO_SINGLE_RW_DATA`	Bit 31-16 = MDI Read data from external deviceBit 15:0 = MDI write data to external device	0x00000000 - 0x0000ffff	32-bits
`GET_MDIO_SINGLE_RW_DATA`	Get MDI read write data	0x00000000 - 0xffffffff	32-bits
`SET_AS_PHY_ADDR`	Set the PHY address enableBit 3:0 = set high to enable PHY address [1]	0x00000000 - 0x0000000f	32-bits
`GET_AS_PHY_ADDR`	Get the PHY address status	0x00000000 - 0x0000000f	32-bits
`SET_MDIO_CTL`	Bit 19:16 = Remote Fault StatusBit 3 = MDI ProgressBit 2 = Set high to enable MDIBit 1 = set high to enable auto-scanBit 0 = select speed of MDC clock	0x00000000 - 0x0000000f	32-bits
`GET_MDIO_CTL`	Get the MDIO Control status	0x00000000 - 0x000f000f	32-bits

2.3.3.8 SPI-3 Configuration ioctl Commands

Table 2-9 lists the `ioctl` commands used to configure and monitor the SPI-3 interface

Table 2-9. ioctl Commands to Configure SPI-3 Interface

SPI-3 Configure Ioctl commands	Description	Range	Buffer size
SET_SPI3_TX_CONFIG	Set the SPI3 Transmitter and Global configuration (4x8 mode)	0x00000000 - 0x00ffffff	32-bits
GET_SPI3_TX_CONFIG	Get the SPI3 Transmitter and Global configuration (4x8 mode)	0x00000000 - 0x00ffffff	32-bits
SET_SPI3_RX_CONFIG	Configure the SPI-3 Receiver	0x00000000 - 0x0fffffff	32-bits
GET_SPI3_RX_CONFIG	Get the SPI-3 Receiver configuration	0x00000000 - 0x0fffffff	32-bits
GET_SPI3_TX_INT_STATUS	Get the status of various SPI-3 transmit error interrupts. (one for each port. [2]	0x00000000 - 0x000001ff	32-bits
GET_SPI3_ADR_PARITY_ERROR	Get the number of packets dropped sue to address parity error.	0x00000000 - 0x000000ff	32-bits
GET_SPI3_PKT_DISABLE_PORT	Get number of packets received for disabled port that has been dropped. [2]	0x00000000 - 0x000000ff	32-bits
GET_SPI3_PKT_SYNC_ERR	Get the number of packets received with full SYNC error (No SOP but EOP) that has been dropped.	0x00000000 - 0x000000ff	32-bits
GET_SPI3_PKT_SHORT_DROP	Get the number of dropped, whose length is less than 9 bytes.	0x00000000 - 0x000000ff	32-bytes

2.3.3.9 SERDES Interface ioctl Commands

Table 2-10 describes the `ioctl` commands used to configure and monitor the SerDes interface.

Table 2-10. ioctl Commands to Configure SERDES Interface (Sheet 1 of 3)

SERDES Interface Ioctl commands	Description	Range	Buffer Size
SET_ACDC_COUPLING	Set AC or DC coupling on the output of each SerDes port (Tx and RX are independent)Bit 7:0 = each pair of bits represents the port number from LSB, and out of that even bit number is for TX and odd is for RX	0x00000000 - 0x000000ff	32-bits
GET_ACDC_COUPLING	Get the AC or DC coupling status	0x00000000 - 0x000000ff	32-bits
SET_SERDES_TX_DRV_COEFF	Set the various programmable strength s on each of the SerDes port	0x00000000 - 0x00ffffff	32-bits
GET_SERDES_TX_DRV_COEFF	Get the strength on each of the SerDes Port	0x00000000 - 0x00ffffff	32-bits
SET_TX_DRV_POW_LEVEL	Set the power level for each of the SerDes port. Each byte corresponds to the port number starting from LSB.	0x00000000 - 0x0000ffff	32-bits
GET_TX_DRV_POW_LEVEL	Get the power level for each port	0x00000000 - 0x0000ffff	32-bits

Table 2-10. ioctl Commands to Configure SERDES Interface (Sheet 2 of 3)

SERDES Interface Ioctl commands	Description	Range	Buffer Size
SET_TX_LINK_VALIDATION	Configure the link status, and stores that.	0x00000000 - 0x00f03cf0	32-bits
GET_TX_LINK_VALIDATION	Get the status of the link validation	0x00000000 - 0x00f03cff	32-bits
SET_TX_RX_POW_DOWN	Set the Power-down TX and RX power-down bits to allow per port power-down of the unused port.Bit 13:10 = Set bit to high to Tx Power down per port (each bit from LSB corresponds to each port number.)Bit 3:0 = set bit high to RX Power-down per port, port mapping to the bit is same as above.	0x00000000 - 0x00003c0f	32-bits
GET_TX_RX_POW_DOWN_STT	Gets port status regarding individuals TX and RX power down.	0x00000000 - 0x00003c0f	32-bits
SET_RX_DATA_SYNC	Enable the feature, which allows the incoming data stream to be slipped by one bit each time the signal is activated.Bit 3:0 = Set bit high to activate the data synchronization control per port. Each bit from LSB maps to the port number.	0x00000000 - 0x0000000f	32-bits
GET_RX_DATA_SYNC	Get the receive data synchronization status.	0x00000000 - 0x0000000f	32-bits
SET_RX_LINK_VALIDATION	Setting these bits allows a BIST test to be carried out to validate the link function.Bit 23:20 = Set high to enable link validation. Each bit from LSB maps to the respective port number.Bit 13:10 = Set high to reset link validation controller. Each bit from LSB maps to the respective port number. Bit 3:0 = set high to enable link validation loop back.	0x00000000 - 0x00f03c0f	32-bits
GET_RX_LINK_VALIDATION	Get the receive validation status.	0x00000000 - 0x00f03c0f	32-bits
GET_RX_LINK_STT	Same as above, but corresponding register is not cleared when read.	0x00000000 - 0x0000000f	32-bits
SET_RX_PHASE_ROT	Control the Phase Rotator in the SerDes Rx on a per port basis.Bit 23:20 = Set high to enable phase rotator retard. Each bit from LSB maps to the respective port numberBit 13:10 = Set high to enable phase rotator. Each bit from LSB maps to the respective port number.Bit 3:0 = set high to enable phase rotator advance. Each bit from LSB maps to the respective port number	0x00000000 - 0x00f03c0f	32-bits
GET_RX_PHASE_ROT	Get the phase rotator status.	0x00000000 - 0x00f03c0f	32-bits
GET_RX_PHASE_ROT_BUS	Get phase rotator state in conjunction with RX phase rotator control.	0x00000000 - 0x00ffffff	32-bits
SET_RX_LATCH_OBSRV_01	Allow the capture of data at the output of the de-serializer SerDes for port 0 and 1.	0x00000000 - 0x00c00801	32-bits

Table 2-10. ioctl Commands to Configure SERDES Interface (Sheet 3 of 3)

SERDES Interface Ioctl commands	Description	Range	Buffer Size
GET_RX_LATCH_OBSRV_01	Get the latch observation for the port 0 and 1.	0x00000000 - 0x00ffffff	32-bits
SET_RX_LATCH_OBSRV_23	Allow the capture of data at the output of the de-serializer SerDes for port 2 and 3.	0x00000000 - 0x00c00801	32-bits
GET_RX_LATCH_OBSRV_23	Get the latch observation for the port 2 and 3.	0x00000000 - 0x00ffffff	32-bits
GET_RX_SIGNAL_LEVEL	Get the status of the Rx input in relation to the level of the signal being received from the line.Bit 3:0 = High bit status depicts Signal, while low for Noise. Each bit from LSB maps to the respective port number	0x00000000 - 0x0000000f	32-bits
GET_CLOCK_INTERFACE_MODE	The register is used to indicate the internal clock generator of when to sample the new value of the interface clock mode (speed) and the interface mode (Copper/Fiber).		32-bits
GET_SERDES_TX_CONFIG	Get the default TX block configuration value.	0x00000000 - 0x000f03c9	32-bits
GET_SERDES_RX_CONFIG	Get the default RX block configuration value	0x00000000 - 0x000f03c9	32-bits
GET_PLL_LOCK	Get the status of the PLL lock for the RX and TX block.	0x00000000 - 0x00000003	32-bits

2.3.3.10 GBIC Interface ioctl Commands

Table 2-11 lists the `ioctl` commands used to control and monitor the GBIC interface

Table 2-11. ioctl Commands to Control and Monitor GBIC Module (Sheet 1 of 2)

GBIC Interface ioctl commands	Description	Range	Buffer Size
GET_GBIC_STAUS	Get the interface status to the GBIC module when used in SerDes mode.	0x00000000 - 0x00f03c0f	32-bits
SET_GBIC_CTL	Configure the GBIC module	0x00000000 - 0x0001fc0f	32-bits
GET_GBIC_CTL	Get the GBIC module configuration	0x00000000 - 0x0001fc0f	32-bits
SET_I2C_CTL_DATA	Set the I2C control data	0x00000000 - 0x013ffe00	32-bits
GET_I2C_CTL_DATA,	Get the I2C control Data	0x00000000 - 0x0d3fffff	32-bits
SET_PLL_TUNE_1	These registers control and adjust the charge pumps, VCO, and internal capacitor tuning of the Serializer/Deserializer blocks, allowing programming for optimal performance in any given system configuration.	0x000003ff	32-bits
GET_PLL_TUNE_1		0x000007ff	32-bits

Table 2-11. ioctl Commands to Control and Monitor GBIC Module (Sheet 2 of 2)

GBIC Interface ioctl commands	Description	Range	Buffer Size
SET_PLL_TUNE_2		0x000003ff	32-bits
GET_PLL_TUNE_2		0x000007ff	32-bits
SET_PLL_TUNE_3		0x000003ff	32-bits
SET_PLL_TUNE_3		0x000007ff	32-bits

2.3.4 Error Types

This section shows error enumerator structure used for returning an error type and list the error types returned by the `GbEMAC_Ioctl()` and `GbEMAC_DeviceStart()` functions.

Error Enumerator

```
typedef enum gbe_mac_e_error {
} gbe_mac_error;
```

2.3.4.1 Error Types from GbEMAC_Ioctl()

Table 2-12 lists the error types returned by the `GbEMAC_Ioctl()` function.

Table 2-12. GbEMAC_Ioctl() Error Types and Description (Sheet 1 of 3)

Error Types	Numeric value	Description
SUCCESS	0x0000	The operation is performed successfully.
IOCLT_ERROR_INPUT_REG_VALUE_OUT_OF_RANGE	0x0405	The input value for register to be set
ERROR_DEVICE_ALREADY_INITIALIZED	0x401	
PLL_NOT_LOCKED	0x402	
ISR_REGISTERTATION_FAILED	0x403	
ISR_DISABLE_FAILED	0x404	
IOCLT_ERROR_INPUT_REG_VALUE_OUT_OF_RANGE	0x405	
IOCLT_ERROR_INPUT_PORT_NUMBER_INVALID	0x0406	Input port number not within [0,3]
IOCTL_ERROR_UNKNOWN_IOCTL_CODE	0x0407	The `ioctl` code is not valid.
IOCTL_BUFFER_POINTER_NULL	0x0408	Passed `ioctl` pointer is NULL.
IOCTL_BUFFER_POINTER_INVALID	0x0409	Passed `ioctl` pointer is invalid.
IOCTL_PORT_NOT_OPEN	0x040A	The port has not been initialized
MODE_ALREADY_SET_IN_SPECIFIED_DUPLEX_MODE	0x040B	The Duplex mode already set in the specified mode.
ERROR_INVALID_DUPLEX_MODE	0x040C	Given mode is invalid.
FAIL_RX_FIFO_ERRORED_FRAME_DROP_IS_DISABLE	0x040D	RX FIFO Error frame drop is disabled.
ERROR_CONFLICT_WITH_HIGH_WATERMARK	0x040E	High watermark level is lesser to low watermark level.
ERROR_VALID_ONLY_FOR_COPPER_MODE	0x040F	The change is valid in only copper mode.
ERROR_VALID_ONLY_FOR_FIBER_MODE	0x0410	The change is valid in fiber mode only.
ERROR_MDI_ENABLE_BIT_IS_RESET_IN_MDI_CONTROL_REG	0x0411	MDI bit in the MDI control register is reset.

Table 2-12. GbEMAC_Ioctl() Error Types and Description (Sheet 2 of 3)

Error Types	Numeric value	Description
ERROR_CONFLICT_WITH_LOW_WATERMARK	0x0412	Low watermark level is lesser to high watermark level.
ERROR_NOT_ALLOWED_IN_LOOPBACK_MODE	0x0413	The change is not allowed in the Loop back mode.
ERROR_CONFLICT_WITH_JUMBO_FRAME_SIZE	0x0414	The defined frame size conflicts with the Jumbo frame size.
PLL_TRANSMIT_LOCK_STATUS_FAILED	0x415	
IOCTL_READ_VALUE_OUT_OF_RANGE	0x0416	Read value is out of range.
ERROR_NPU_IS_INGRESS	0x417	
PCI_ERROR_COULD_NOT_GRAB_SEMAPHORE	0x418	
ERROR_MAC_NOT_INITIALIZED	0x0419	MAC had not been initialized earlier.
ERROR_GET_RMON_STAT_TASK_COULD_NOT_SPAWNED	0x041A	Task for gathering the RMON statistics could not be spawned.
MEMORY_ALLOCATION_FAILS	0x041B	Memory could not be allocated.
ERROR_INT_LOCK_FAILED	0x041C	Interrupt Lock failed.
ERROR_TASK_LOCK_FAILED	0x041D	Task Lock failed
SEMAPHORE_COULD_NOT_CREATED	0x041E	Semaphore could not be created.
ERROR_PT_LONE_MEDIA_CARD_IS_NOT_PRESENT_ON_MEDIA_INTERFACE	0x41f	Pt. Lone card is not present on the Media interface.
ERROR_PT_LONE_MEDIA_CARD_IS_NOT_PRESENT_ON_SWITCH_FABRIC_INTERFACE	0x0420	Pt. Lone media card is not present on the switch fabric interface.
EEPROM_OF_PT_LONE_MEDIA_CARD_ON_MEDIA_INTERFACE_IS_NOT_PROGRAMMED	0x0421	EEPROM on the Pt. lone media card on the MEDIA interface is not programmed.
EEPROM_OF_PT_LONE_MEDIA_CARD_ON_SWITCH_FABRIC_INTERFACE_IS_NOT_PROGRAMMED	0x0422	EEPROM on the Pt. lone media card on the Switch Fabric interface is not programmed.
ERROR_INVALID_HANDLE	0x0423	Invalid handle passed by the application.
ERROR_PREVIOUS_MDI_COMMAND_STILL_NOT_COMPLETED	0x0424	Previous MDI command still not completed.
ERROR_COULD_NOT_PERFORM_MDIO_READ	0x0425	MDIO read fails
ERROR_MDI_RD_WR_DISABLE	0x0426	MDIO single read-write operation disables.
ERROR_COULD_NOT_PERFORM_MDIO_WRITE	0x0427	MDIO write fails
SPI3_TX_LOCK_FAILED	0x428	
NO_MORE_CALLBACK_FOR_INTERRUPT	0x429	
CALLBACK_OVER_WRITTEN	0x42a	

Table 2-12. GbEMAC_Ioctl() Error Types and Description (Sheet 3 of 3)

Error Types	Numeric value	Description
CALLBACK_REGISTERED	0x42b	
INTERRUPT_TYPE_COULD_NOT_RECOGNIZED	0x42c	
WARNING_ONLY_LOWER_FOUR_BYTES_ARE_WRITABLE	0x0430	
WARNING_ONLY_LOWER_THREE_BITS_ARE_WRITABLE	0x0431	

2.3.4.2 Error Types from GbEMAC_DeviceStart()

Table 2-13 lists the error types returned by the `GbEMAC_DeviceStart()` function.

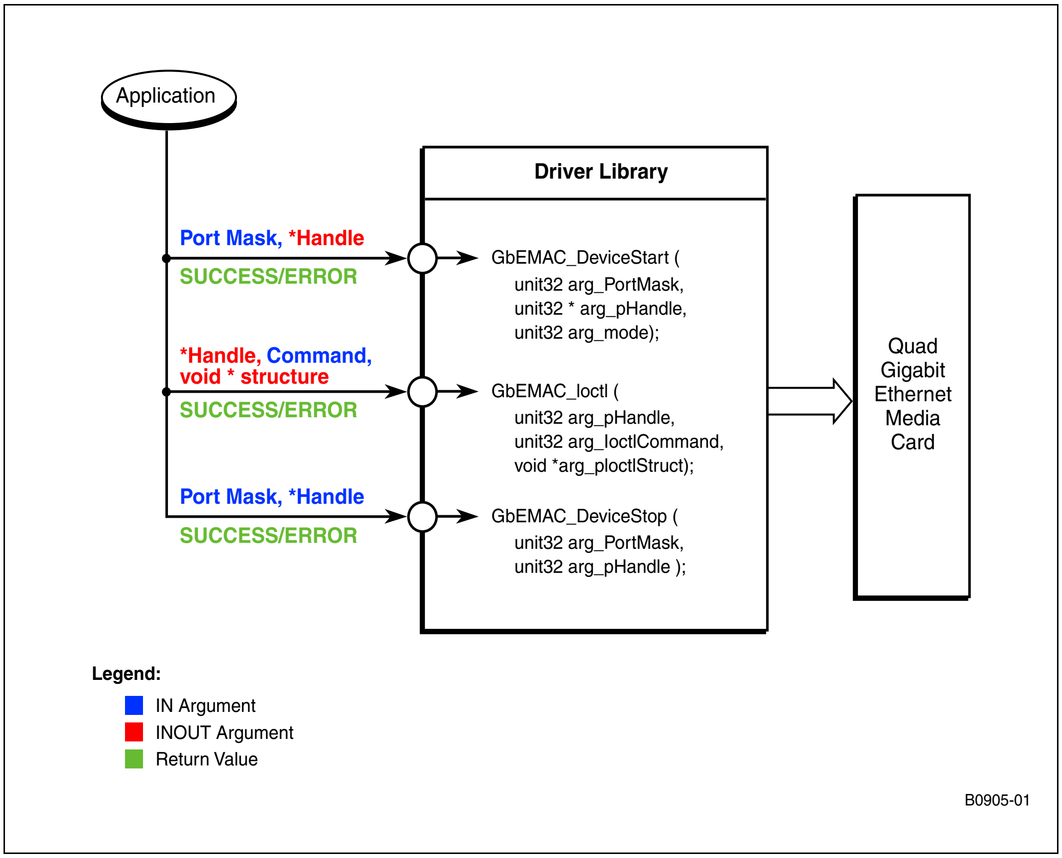
Table 2-13. GbEMAC_DeviceStart() Error Types and Description

Error Types	Numeric Value	Description
ERROR_DEVICE_ALREADY_INITIALIZED	0x0401	Device is already initialized.
PLL_NOT_LOCKED	0x0402	PLL lock not achieved
ISR_REGISTERTATION_FAILED	0x0403	pciIntConnect() failed
ISR_DISABLE_FAILED	0x0404	pciIntDisconnect () failed
PLL_TRANSMIT_LOCK_STATUS_FAILED	0x0415	PLL lock has not been achieved

2.4 API Usage Model

This usage model describes the usage of the APIs, exported by the driver library, for initializing and configuring the device. The driver library supports multiple devices.

Figure 2-2. API Driver Library



The driver APIs interface with the calling application to configure and initialize the Quad GbE I/O card. The APIs are defined in the `ixf1104ce_driver_api.h` file and the driver API header files in `ixf1104ce_driver_api.h`.

The calling application interfaces with the driver API library to open the multiple ports of interest by passing the port mask to indicate the ports to be opened. On successful completion of this open call, the application calls an API to perform the change configuration using the IOCTL command. When the calling application exits, it calls an API to close the ports it has opened earlier.

2.5 VxWorks Driver APIs

This section defines the ways in which the application interfaces with the driver APIs.The Quad GbE I/O card driver module provides interfaces for the application to access the I/O card registers.

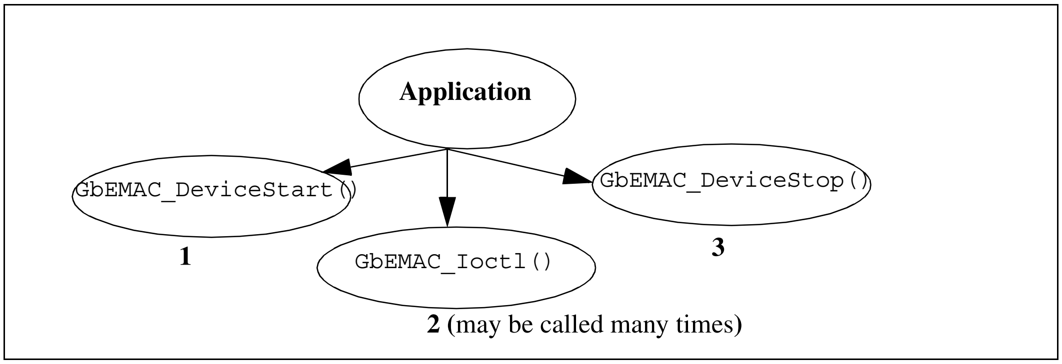
Table 2-14 shows the I/O Card driver APIs.

Table 2-14. GbE MAC I/O Card Driver APIs

API	Description
`GbEMAC_DeviceStart()`	Called by the application to open the device
`GbEMAC_DeviceStop()`	Called by the application to close the driver
`GbEMAC_Ioctl`	Called by the application to run an `ioctl` command

Figure 2-3 illustrates the calling sequence of each function.

Figure 2-3. Function Calling Sequence



2.5.1 GbEMAC_DeviceStart()

This routine is called by the application to open the device. The `GbEMAC_DeviceStart()` function performs the following tasks:

- Initializes the port, whose mask is passed as an argument to this function and configures the device
- Configures the device in the specified mode. If an illegal mode is specified, the configuration defaults to the fiber mode.
- Sets the rest of the registers to their default configuration values—initializes the global and per port registers of the device
- Sets the port state as opened

After initializing and configuring the registers, it calls the interrupt routine to connect the interrupt handler function to the interrupt vector.

Once the port has been initialized by the calling application, any subsequent call to this API would return SUCCESS without re-initializing the port again. This function should not be called again without first calling the `GbEMAC_DeviceStop()` function.

Note: The header files required to be included in the application code are:

- `#include "VxWorks/include/ixf1104ce_driver_api.h"`
- `#include "common/include/ixf1104ce_ioctl.h"`

Syntax

```
uint32 GbEMAC_DeviceStart (uint32 arg_PortMask
                           uint32 * arg_pHandle,
                           uint32 arg_mode);
```

Input

`arg_PortMask`	This mask indicates the ports to be opened—0x00-0xFF
`arg_pHandle`	The driver library writes a handle for the application.
`arg_mode`	Specifies the mode in which the port needs to be opened. The ports represented by the `arg_PortMask` are opened in the specified mode by the argument `arg_PortMode`. See Table 2-15 for the specified mode value interpretation.

Table 2-15. Mode Value Interpretation

Bit Position	Associated to	Value Intepretation
4	SPI-3 Block Mode	0 - SPHY 4x8 Mode 1 - MPHY Mode
3	SPI-3 Parity	0 - Odd Parity 1 - Even Parity
2-0	Channel, duplex and speed selection mode	000 Fiber Mode 001 Copper 1000 Half Duplex Copper 010 Copper 1000 Full Duplex Copper 011 Copper 100 Half Duplex Copper 100 Copper 100 Full Duplex Copper 101 Copper 10 Half Duplex Copper 110 Copper 10 Full Duplex Copper

Output/Returns

Return Type • SUCCESS or a valid `gbe_mac_error` type

2.5.2 GbEMAC_DeviceStop()

This function is called by the calling application to close the device in use. This function resets all the GbE MAC media card configuration registers to zero and sets the port state as CLOSED.

The calling application passes the address of the handle received from the `GbEMAC_DeviceStart()` API and this function checks the handle for validity. If the calling application is the last application using the driver of the specified port, then it sets the port status as CLOSED and the port is no longer in active state. For using the port again, the application must call the `GbEMAC_DeviceStart()` API.

If the calling application is not the last application, it decrements the usage count for that device, and sets the handle value to zero indicating that the application is no longer interested in using the driver. This function should be called after the `GbEMAC_DeviceStart()` has been successfully called.

Syntax

```
uint32 GbEMAC_DeviceStop (
    uint32 arg_PortMask,
    uint32 * arg_pHandle);
```

Input

`arg_PortMask`	An 8-bit value that represents the ports to be closed. Each high bit indicates the port by its position from LSB. [0x00-0xFF]
`arg_pHandle`	The pointer to the handle received from the `GbEMAC_DeviceStart()` API and is used to verify the application identity.

Output/Returns

Return Type	• SUCCESS or a valid `gbe_mac_error` type

2.5.3 GbEMAC_Ioctl

This routine is provided to call an `ioctl`. This function is the entry point to configure and get status of the device. This routine performs different functions based upon the function parameter. This routine calls the `gbe_mac_config_handler` routine for the implementation of the `ioctl` command.

This function can only be called after the `GbEMAC_DeviceStart()` function has been successfully called and it ensures validity the calling application by checking the handle returned by `GbEMAC_DeviceStart()` function.

Syntax

```
extern uint32 GbEMAC_Ioctl (
    uint32 * arg_pHandle,
    uint32 arg_IoctlCommand,
    void *arg_pIoctlStruct);
```

Input

`arg_HandleID`	A unique handle returned by `GbEMAC_DeviceStart()`.
`arg_ioctlCommand`	The `ioctl` command which is to be performed. The corresponding `ioctl` number is parsed in the driver which performs the `ioctl` operation

Input

`arg_pIoctlStruct`	The `ioctl` structure pointer that contains the port number and integer pointer. The port number specifies the MAC port to which this `ioctl` is intended and the `uint32 *` points to the value to be written to the particular resistor for `SET ioctl` commands and stores the value read in `GET ioctl` commands.

Output/Returns

Return Type	• SUCCESS or a valid `gbe_mac_error` type

2.5.4 GbEMAC_Callback()

This function is called by the calling application to register the callback function in the driver and the driver library calls this function when an interrupt is generated.

The calling application passes the address of the handle received from the `GbEMAC_DeviceStart()` API and this function checks the handler for validity of the calling application, and it is used in removing the associated callback functions in the `GbEMAC_DeviceStop()`.

Syntax

```
gbe_mac_error GbEMAC_Callback(
    uint32 *arg_Handle,
    VOIDFUNCPTR *arg_pCallback,
    VOID* arg_pUserContext
);
```

Input

`arg_Handle:`	A unique handle returned by `GbEMAC_DeviceStart()`.
`*arg_pCallback`	Pointer to the user application function, which is passed to this API to register with the driver as a callback function.
`arg_pUserContext`	Pointer to the user context passed as a parameter to the callback function.

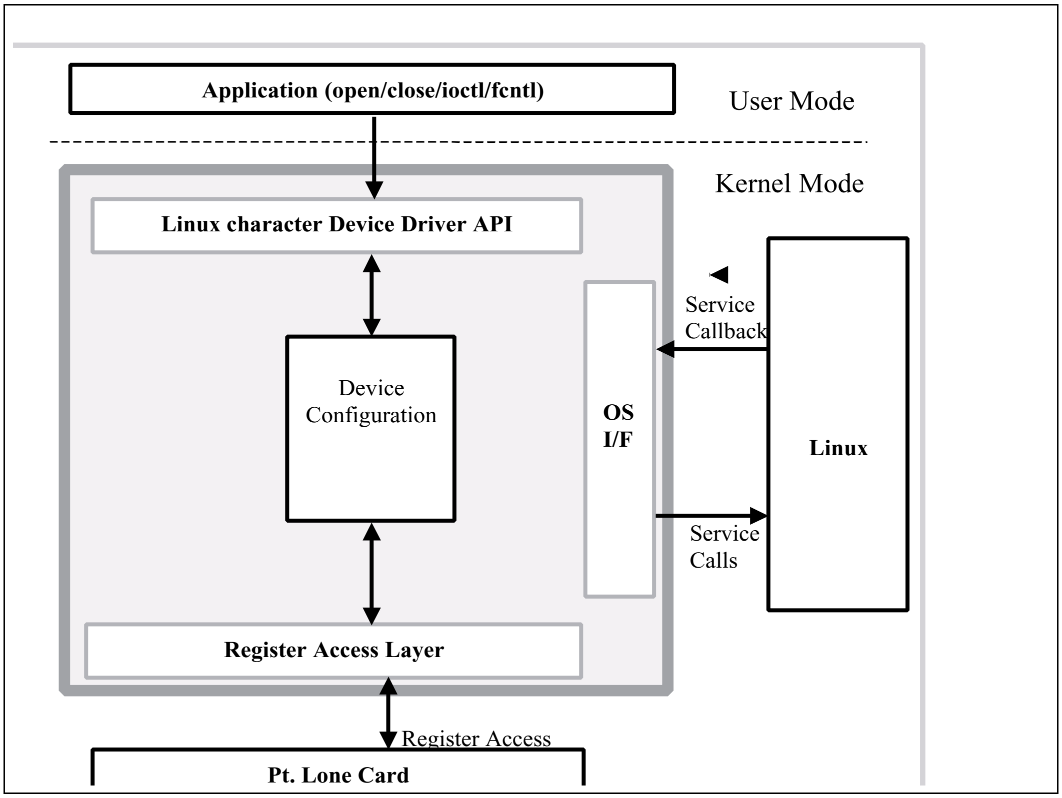
Output/Returns

Return Type	• SUCCESS—the callback function pointer has been successfully registered • A valid `gbe_mac_error` type

2.6 Linux Environment

The following figure shows an overview of the device driver architecture for the Linux platform. The figure shows the environment in which the driver is to execute, the major components of the design, and relationship among the components.

Figure 2-4. Linux Driver Architecture



2.6.1 Linux Character Device Driver APIs

The driver API is a set of high-level functions that are invoked by applications needing to initialize and configure the Quad Gigabit Ethernet I/O card. This driver is modeled as a character driver for Linux platform. The API includes the following function types:

Function Type	Description
Init	Initializes and configures the device.
Ioctl handlers	Performs device configuration and provides status.
Open	Updates the status of the port to be opened and activated.
Close	Sets the status of the device to ‘deactivate’ so the port will no longer be in normal operation.
Fcntl	Gets the asynchronous notification (fasync) of the interrupt in form of the signal.

2.6.2 Device Configuration

The device configuration encompasses the initialization and configuration functions. The ioctls are provided to alter the basic configuration in the desired mode.

2.6.3 Operating System Interface

The driver's operating system interface provides functions that let the driver use the operating system services related to interrupt handling and memory access. This is to ensure the portability and code reuse.

2.6.4 Register access layer

The register access layer provides functions/macros that read from and write to the device registers.

2.7 Data Structures

Data structures incorporate the list of `ioctl` commands, used by the user application to configure the device. The `ioctl` list describes the input/output control commands for configuring various registers and to get their status. The error enumerator maintains the error number, which is returned by the driver module, in case an error condition occurs. The unique error message corresponding to each error number describes the nature of the error. The port status enumerator maintains the per port status of all the Vallejo MAC on the Quad Gigabit Ethernet I/O Card.

2.7.1 Basic Data Type

Table defines the basic data types that are used in the driver. These types are defined to ease portability of the code across different operating systems.

Table 2-16. Basic Data Type

Basic Types	Description
uint32	32 bit unsigned integer

2.7.1.1 PORT Status

This example provides the different possible states a port can be in.

```
typedef enum gbe_mac_e_port_state {
    CLOSED = 0,
    OPENED
} gbe_mac_port_state;
```

2.7.2 Structure Passed to ioctl Command

The API provided for calling the `ioctl` command contains a `void` pointer as one of its argument. The calling application passes a structure pointer, which maintains the information regarding the `ioctl` to be called. This structure is passed after typecasting it with the `void *` pointer. This structure, whose pointer is passed by the calling application while using the `ioctl` command, is defined below.

The `ioctl` command normally deals with a single register, that is, 32-bits, and some of the specific `ioctl` commands need to read two, 32-bits registers. Structure definition for a single 32-bit register and two 32-bits registers are described below.

ioctl command structure definition for a single register

```
typedef struct gbe_mac_s_ioctl_ptr {
    uint32 portIndex;
    uint32 value;
} gbe_mac_iotcl_ptr;
```

Input

`portIndex`	Indicates the port number
`value`	A placeholder for a value, which is either to be set or retrieved in an `ioctl` operation.

ioctl command structure definition for two registers

```
typedef struct gbe_mac_s_ioctl_ptr_64 {
    uint32 portIndex;
    uint32 valueHigh;
    uint32 valueLow;
} gbe_mac_iotcl_ptr_64;
```

Note: This structure definition shown for two registers is used in the `GET ioctl` commands such as `GET_STN_ADDR`, `GET_FDFC_ADDR`, and `GET_MUL_PORT_ADD`, where the 48-bit data for the MAC address is to be read.

2.7.3 GbE_MAC_ERROR Enumerator

The error codes, returned by the driver code, are described in the following tables. The `IOCTL` Error Code table describes the error codes returned by the `IOCTL` API, while the Initialiazation Error Code table describes the errors returned by the open API.

Table 2-17. `IOCTL` Error Code (Sheet 1 of 3)

Error Code	Numeric Value	Description
SUCCESS	0x0000	The operation is performed successfully.
IOCLT_ERROR_INPUT_REG_VALUE_OUT_OF_RANGE,	0x0405	The input value for register to be set
IOCLT_ERROR_INPUT_PORT_NUMBER_INVALID,	0x0406	Input port number not within [0,3]
IOCTL_ERROR_UNKNOWN_IOCTL_CODE,	0x0407	The ioctl code is not valid.
IOCTL_BUFFER_POINTER_NULL,	0x0408	Passed ioctl pointer is NULL.
IOCTL_BUFFER_POINTER_INVALID,	0x0409	Passed ioctl pointer is invalid.
IOCTL_PORT_NOT_OPEN,	0x040A	The port has not been initialized
MODE_ALREADY_SET_IN_SPECIFIED_DUPLEX_MODE,	0x040B	The Duplex mode already set in the specified mode.
ERROR_INVALID_DUPLEX_MODE,	0x040C	Given mode is invalid.
FAIL_RX_FIFO_ERRORED_FRAME_DROP_IS_DISABLE,	0x040D	RX FIFO Errored frame drop is disabled.
ERROR_CONFLICT_WITH_HIGH_WATERMARK,	0x040E	High watermark level is lesser to low watermark level.
ERROR_VALID_ONLY_FOR_COPPER_MODE,	0x040F	The change is valid in only copper mode.
ERROR_VALID_ONLY_FOR_FIBER_MODE,	0x0410	The change is valid in fiber mode only.
ERROR_MDI_ENABLE_BIT_IS_RESET_IN_MDI_CONTROL_REG,	0x0411	MDI bit is reset in the MDI control register.
ERROR_CONFLICT_WITH_LOW_WATERMARK,	0x0412	Low watermark level is lesser to high watermark level.
ERROR_NOT_ALLOWED_IN_LOOPBACK_MODE,	0x0413	The change is not allowed in the Loopback mode.
ERROR_CONFLICT_WITH_JUMBO_FRAME_SIZE,	0x0414	The packet size conflicts with the jumbo packet size.
PLL_TRANSMIT_LOCK_STATUS_FAILED	0x0415	PLL lock has not been achieved
IOCTL_READ_VALUE_OUT_OF_RANGE,	0x0416	The IOCTL Value read, is out of range.
ERROR_COULD_NOT_COPY_RMON_STAT_FROM_KERNEL_MODE	0x0417	The memcpy_tofs fails, as it could not copy the contents of the kernel space to the specified user case.
ERROR_MAC_NOT_INITIALIZED,	0x0418	MAC hasn't been initialized yet.
ERROR_GET_RMON_STAT_TASK_COULD_NOT_SPAWNED,	0x0419	Task for gathering the RMON statistics could not be spawned.

Table 2-17. IOCTL Error Code (Sheet 2 of 3)

Error Code	Numeric Value	Description
ERROR_COULE_NOT_OPEN_DEVICE,	0x041a	Could not the open the character device file.
SEMAPHORE_COULD_NOT_CREATED	0x041b	Could not create the semaphore
ERROR_PT_LONE_MEDIA_CARD_IS_NOT_PRESENT_ON_MEDIA_INTERFACE,	0x041c	Pt Lone card is not present on the Media interface.
ERROR_PT_LONE_MEDIA_CARD_IS_NOT_PRESENT_ON_SWITCH_FABRIC_INTERFACE,	0x041d	Pt Lone media card is not present on the switch fabric interface.
EEPROM_OF_PT_LONE_MEDIA_CARD_ON_MEDIA_INTERFACE_IS_NOT_PROGRAMMED,	0x041e	EEPROM on the Pt lone media card on the MEDIA interface is not programmed.
EEPROM_OF_PT_LONE_MEDIA_CARD_ON_SWITCH_FABRIC_INTERFACE_IS_NOT_PROGRAMMED,	0x041f	EEPROM on the Pt lone media card on the Switch Fabric interface is not programmed.
ERROR_INVALID_HANDLE,	0x0420	Invalid handle passed by the application.
ERROR_PREVIOUS_MDI_COMMAND_STILL_NOT_COMPLETED,	0x0421	Previous MDI command still not completed.
ERROR_COULD_NOT_PERFORM_MDIO_READ	0x0422	MDIO read operation could not performed successfully
ERROR_MDI_RD_WR_DISABLE	0x0423	Could not perform MDIO read/write operation , as the Rd/WR bit in the MDI register is disabled.
ERROR_COULD_NOT_PERFORM_MDIO_WRITE	0x0424	MDIO write operation could not performed successfully
SPI3_TX_LOCK_FAILED	0x0425	SPI3 transmit lock failed.
ERROR_SHARED_MEMORY_NOT_CREATED	0x0426	Shared memory for holding the global data in the user mode driver could not be created.
ERROR_SHARED_MEMORY_NOT_LOCKED	0x0427	Shared memory could not lock for exclusive access of the data.
ERROR_SHARED_MEMORY_NOT_UNLOCKED	0x0428	Shared memory could not be unlocked.
ERROR_SHARED_MEMORY_NOT_DESTROYED	0x0429	Shared memory couldn't be destroyed.
ERROR_SHARED_MEMORY_NOT_DETATCHED	0x042a	Shared memory could not be detached from the application to which it has been attached.
ERROR_SHARED_MEMORY_NOT_ATTACHED	0x042b	Shared memory could not be attached.
ERROR_SEMAPHORE_NOT_GRABED	0x042c	Semaphore could not be grabbed.
INVALID_PORT_MASK	0x042d	The port mask passed to the driver library is invalid.
MEMORY_ALLOCATION_FAILS	0x042e	Memory could not be allocated for the specified task.
ERROR_SP_CSR_BASE_COULD_NOT_MAPPED	0x042f	Slow port registers could not be mapped, using mmap system call.
WARNING_ONLY_LOWER_FOUR_BYTES_ARE_WRITABLE	0x0430	Only bits 0-15 are writable, rest are reserved.
WARNING_ONLY_LOWER_THREE_BITS_ARE_WRITABLE	0x0431	Only bits 0-2 are writable, rest are reserved.

Table 2-17. `IOCTL` Error Code (Sheet 3 of 3)

Error Code	Numeric Value	Description
NO_MORE_CALLBACK_FOR_INTERRUPT	0x0432	Number of callback functions supported by the driver are full.
ERROR_PORT_ALREADY_OPENED	0x0440	Specified port is already opened.
ERROR_INPUT_PORT_NUMBER_INVALID	0x0441	Passed input port number is invalid.
ERROR_DEVICE_NOT_OPENED	0x0442	Device could not be opened.
ERROR_PORT_NOT_OPEN	0x0443	Specified port is still not opened.
ERROR_CALLED_IOCTL_FAILED	0x0444	Called Ioctl has encountered an error.

Table 2-18. Initialization Error Code

Error code	Numeric Value	Description
`REGISTRATION_FAILED`	0x0400	GbEMAC Module can not be registered with the kernel.
`ERROR_DEVICE_ALREADY_INITIALIZED`	0x0401	Device is already initialized.
`PLL_NOT_LOCKED`	0x0402	PLL lock not achieved
`INTERRUPT_INSTALLATION_FAILED`	0x0403	ISR could not be registered with the interrupt vector.
`INTERRUPT_FREEING_FAILED`	0x0404	ISR could not be unregistered from the interrupt vector.

The application calling an `IOCTL` function expects the return value, in case an error is returned it uses the returned value to map the error enumerator to get the error message. The returned value can be used as an index to the array of the error messages.

2.7.4 IOCTL_CMD Enumerator

The `IOCTL_CMD` enumerator defines various `ioctls` used by the application. The application passes `ioctl` command code as one of the arguments in the `ioctl()` system call. The registers of the device are set by using the “SET” `ioctl` commands. The “SET” prefixes are used for setting the register to the given parameter. The “GET” prefixes are the `ioctl` commands used to get the register value. The `ioctl` command list is provided in the following tables.

2.7.4.1 MAC Control Ioctls

Table 2-19 explains the `ioctls`[1] used for configuring and monitoring the status of the registers associated with each MAC port.

1. The Port Number should be given separately, as a parameter to the buffer, whose pointer is passed as an argument to the ioctl command

Table 2-19. MAC Control IOCTL Commands (Sheet 1 of 3)

IOCTL Command MAC control ioctls	Description	Defined Value	Buffer Size
SET_STN_ADDR_LOW	Set source MAC address bit 31-0	0xe200	32-bits
SET_STN_ADDR_HIGH	Set source MAC address bit 47-0	0xe201	32-bits
GET_STN_ADDR	Get source MAC address	0xe300	64-bits
SET_DUPLEX_MODE	Set half/full-duplex operation mode of the MAC	0xe202	32-bits
GET_DUPLEX_MODE	Get the MAC operating mode	0xe302	32-bits
SET_FDFC_TYPE	Set FDFC Type field of the Transmit Pause Frame	0xe203	32-bits
GET_FDFC_TYPE	Get FDFC Type field of the Transmit Pause Frame	0xe303	32-bits
SET_COLLISION_DIST	Set limit for the late collisions	0xe204	32-bits
GET_COLLISION_DIST	Get the limit set for late collision	0xe304	32-bits
SET_COLLISION_THLD	Set the limit for excessive collision	0xe205	32-bits
GET_COLLISION_THLD	Get the limit set for excessive collision	0xe305	32-bits
SET_FCTX_TIMER	Set the pause length sent to the receiving station	0xe206	32-bits
GET_FCTX_TIMER	Get the pause length set	0xe306	32-bits
SET_FDFC_ADDR_LOW	Set 31-0 bits of the 48-bit globally assigned multicast pause frame destination address	0xe207	32-bits
SET_FDFC_ADDR_HIGH	Set 47-32 bits of the 48-bit globally assigned multicast pause frame destination address	0xe208	32-bits
GET_FDFC_ADDR	Get the Multicast pause frame destination MAC address	0xe307	64-bits
SET_IPG_RECEIVE_TIME1	Set the first part of the IPG time for non back-to-back transmission	0xe209	32-bits
SET_IPG_RECEIVE_TIME2	Set the second part of the IPG time for non back-to-back transmission	0xe20a	32-bits
GET_IPG_RECEIVE_TIME	Get the IPG time for non back-to-back transmission	0xe309	32-bits
SET_IPG_TRANSMIT_TIME	Configure IPG time for back-to-back transmission	0xe20b	32-bits
GET_IPG_TRANSMIT_TIME	Get IPG for back-to-back transmission	0xe30b	32-bits
SET_PAUSE_THRESHOLD	Set the time between two consecutive pause frames to keep the link partner in pause mode.	0xe20c	32-bits
GET_PAUSE_THRESHOLD	Get the pause threshold time	0xe30c	32-bits
SET_MAX_FRAME_SIZE	Set the maximum frame size the MAC can receive and transmit without activating any error.	0xe20d	32-bits
GET_MAX_FRAME_SIZE	Get the maximum frame size	0xe30d	32-bits
SET_MAC_IF_MODE	Set the MAC operation frequency and mode per port	0xe20e	32-bits
GET_MAC_IF_MODE	Get the MAC operation frequency and mode	0xe30e	32-bits
SET_FLUSH_TX	Set this bit to flush all transmit data. It is set if all the traffic to a port should be stopped.	0xe20f	32-bits
GET_FLUSH_TX	Get the status of this bit,	0xe30f	32-bits
SET_FC_MODE	Set the flow control mode for the RX and TX MAC	0xe210	32-bits
GET_FC_MODE	Get the flow control mode of the RX and TX MAC	0xe310	32-bits

Table 2-19. MAC Control IOCTL Commands (Sheet 2 of 3)

SET_FC_BACK_PRESSURE_LEN	Set the minimum length/duration of backpressure. These six bits holds the value in bytes.	0xe211	32-bits
GET_FC_BACK_PRESSURE_LEN	Get the minimum length/duration of the backpressure	0xe311	32-bits
SET_SHORT_RUNT_TH	Set the threshold to determine between short and runt. The 5-bit value holds the value in bytes.	0xe212	32-bits
GET_SHORT_RUNT_TH	Get the threshold set for the demarcation between short and runt	0xe312	32-bits
SET_UNKNOWN_FRAME_STT	Used to discard/keep the unknown control frames. Known control frames are pause frames.	0xe214	32-bits
GET_UNKNOWN_FRAME_STT	Check the action regarding the unknown frames	0xe314	32-bits
GET_RX_CONFIG_WORD	This is used in Fiber MAC only for auto negotiation. The contents of this register are the “config_word” received from the link partner	0xe315	32-bits
SET_TX_CONFIG_WORD	Set this register which is used in Fiber MAC for auto-negotiation only. The contents of this register are sent as the config_word.	0xe216	32-bits
GET_TX_CONFIG_WORD	Get the config_word register contents, sent for auto-negotiation.	0xe316	32-bits
SET_DIV_CONFIG_WORD	Set various configuration bits for general use.	0xe217	32-bits
GET_DIV_CONFIG_WORD	Get the various configuration status	0xe317	32-bits
SET_PKT_FILTER_CTL	Set this register to allow specific packet types to be marked for filtering. This is used in conjunction with the RX FIFO errored Frames Drop Enable Register	0xe218	32-bits
GET_PKT_FILTER_CTL	Get the status regarding the packet filtering	0xe318	32-bits
SET_MUL_PORT_ADD_LOW	Set bit 31:0 of the address. This address is used to compare against multicast frames at the receiving side if multicast filtering is enabled.	0xe219	32-bits
SET_MUL_PORT_ADD_HIGH	Set bit 47:32 of the address	0xe21a	32-bits
GET_MUL_PORT_ADD		0xe319	64-bits
SET_PHY_REGISTER	Set the PHY register value Bit 0-15 contains the value written to the register, and Bit 16-20 represents the PHY register number for the specified port.	0xe244	32-bits
GET_PHY_REGISTER	Get the PHY register value In the value, the PHY register number is passed.	0xe398	32-bits

Table 2-19. MAC Control IOCTL Commands (Sheet 3 of 3)

SET_CONFIG_MODE	Change the port configuration value, e.g. from port configuration Fiber to Copper, or link speed, or mode etc. The interpretation of the 0-5 bits for this ioctls is as follows, Bit Value Description 4 1 SPHY 4x8 Mode 0 MPHY Mode 3 0 ODD Parity 1 EVEN Parity 0 – 2 000 Fiber Mode 001 1000 Half Duplex Copper 010 1000 Full Duplex Copper 011 100 Half Duplex Copper 100 100 Full Duplex Copper 101 10 Half Duplex Copper 110 10 Full Duplex Copper Ioctl Command is 0xe245.		32-bits
SET_PORT_INIT_ENABLE	This particular IOCTL is called from the open routine in the kernel mode driver, to increment the soft count of the port.	0xe242	32-bits
SET_PORT_STATE_CLOSE	This particular IOCTL is called from the close routine in the kernel mode driver, to decrement/update the soft count of the port.	0xe243	32-bits

2.7.4.2 MAC Receive Control Ioctls

Table 2-20 presents the ioctls which are used to monitor the MAC Receive Statistics counters contents. These ioctls can be used in polling. These registers are cleared when read. When RX statistics counter overflows, it gets wrapped back to zero. At the Gbps speed, the 32-bit counters wrap after approximately 30 seconds. The driver polls these registers and accumulates values in virtual 64-bit counters (2-32 bit registers) to ensure that the RX statistics counters do not wrap. So for these IOCTLs the user needs to pass 2-32bit registers to get the 64-bit register value.

Table 2-20. MAC Receive Statistics Counters ioctls Commands (Sheet 1 of 2)

IOCTL Commands MAC RX Stat ioctls	Description	Defined Value	Buffer Size
GET_RX_OCTETS_OK	Get the number of bytes received in all legal frames, including all bytes from the destination MAC address to and including the CRC. The initial preamble and SFD bytes are not counted.	0xe31a	32-bits
GET_RX_OCTETS_BAD	Get the number of bytes received in all bad frames with legal size	0xe31b	32-bits
GET_RX_UC_PKTS	Get the total number of unicast packets received, (EBP)	0xe31c	32-bits
GET_RX_MC_PKTS	Get the total number of multicast packets received (EBP)	0xe31d	32-bits
GET_RX_BC_PKTS	Get the total number of broadcast packets received (EBP)	0xe31e	32-bits
GET_RX_PKTS_64	Get the total number of packets received (IBP) that are 64 octets in length	0xe31f	32-bits
GET_RX_PKTS_65_127	Get the total number of packets received (IBP) that are [65-127] octets in length.	0xe320	32-bits
GET_RX_PKTS_128_255	Get the total number of packets received (IBP) that are [128-255] octets in length	0xe321	32-bits

Table 2-20. MAC Receive Statistics Counters ioctls Commands (Sheet 2 of 2)

GET_RX_PKTS_256_511	Get the total number of packets received (IBP) that are [256-511] octets in length.	0xe322	32-bits
GET_RX_PKTS_512_1023	Get the total number of packets received (IBP) that are [512-1023] octets in length	0xe323	32-bits
GET_RX_PKTS_1024_1518	Get the total number of packets received (IBP) that are [1024-1518] octets in length	0xe324	32-bits
GET_RX_PKTS_1519_MAX	Get the total number of packets received (IBP) that are >1518 octets in length.	0xe325	32-bits
GET_RX_FCS_ERR	Get the number of frames, received with legal size, but with wrong CRC field (also called FCS field).	0xe326	32-bits
GET_VLAN_TAG	Get the number of OK frames with VLAN tag	0xe327	32-bits
GET_RX_DATA_ERR	Get the number of frames, received with the legal length with code violation.	0xe328	32-bits
GET_RX_ALLIGN_ERR	Get the number of frames, with a legal frame size, but containing less than 8 additional bits	0xe329	32-bits
GET_RX_LONG_ERR	Get the number of frames, bigger than the maximum allowed, with both OK CRC and the integral number of octets.	0xe32a	32-bits
GET_RX_JABBER_ERR	Get the number of frames, bigger than the maximum allowed, with either a bad CRC or a non-integral number of octets	0xe32b	32-bits
GET_RX_PAUSE_MAC_CTL	Get the number of Pause MAC control frames received	0xe32c	32-bits
GET_RX_UNKNOWN_CTL_FRAME	Get the number of MAC control frames, received with an op code different from 0001 (Pause)	0xe32d	32-bits
GET_VLONG_ERR	Get the number of frames, bigger than the larger of 2*maxframesize and 50000 bits	0xe32e	32-bits
GET_RUNT_ERR	Get the total number of packets, received that are less than 64 octets in length, but longer than or equal to 96 bit times, which corresponds to a 4- byte frame with a well formed preamble and SFD	0xe32f	32-bits
GET_SHORT_ERR	Get the total number of packets, received that are less than 96 bit times, which corresponds to a 4- byte frame with a well formed preamble and SFD.	0xe330	32-bits
GET_SEQ_ERR	Get the number of sequencing errors that occur in Fiber mode.	0xe331	32-bits
GET_SYMBOL_ERR	Get the number of symbol errors, encountered by the PHY	0xe332	32-bits

2.7.4.3 MAX Transmit Control Ioctls

The MAC Transmit Statistics Counters ioctls table describes the `ioctl` commands used to monitor the MAC Transmit Statistics counters contents. These commands can be used in polling. The corresponding registers are cleared when read . When TX statistics counter overflows, it gets wrapped back to zero. At the Gbps speed, the 32-bit counters wrap after approximately 30 seconds. The driver polls these registers and accumulates values in virtual 64-bit counters (2-32 bit registers) to ensure that the RX statistics counters do not wrap. For these `ioctl` commands the application must pass two 32-bit registers to get the 64-bit register value.

Table 2-21. MAC Transmit Statistics Counters ioctls (Sheet 1 of 2)

IOCTL Commands MAC TX Stat ioctls	Description	Defined Value	Buffer Size
GET_TX_OCTETS_OK	Get the number of bytes transmitted in all legal frames	0xe333	32-bits
GET_TX_OCTETS_BAD	Get the number of bytes transmitted in all bad frames.	0xe334	32-bits
GET_TX_UC_PKTS	Get the total number of unicast packets transmitted. (EBP)	0xe335	32-bits
GET_TX_MC_PKTS	Get the total number of multicast packets transmitted. (EBP)	0xe336	32-bits
GET_TX_BC_PKTS	Get the total number of broadcast packets transmitted. (EBP)	0xe337	32-bits
GET_TX_PKTS_64	Get the total number of packets transmitted (IBP) that are 64 octets in length	0xe338	32-bits
GET_TX_PKTS_65_127	Get the total number of packets transmitted (IBP) that are [65-127] octets in length	0xe339	32-bits
GET_TX_PKTS_128_255	Get the total number of packets transmitted (IBP) that are [128-255] octets in length	0xe33a	32-bits
GET_TX_PKTS_256_511	Get the total number of packets transmitted (IBP) that are [256-511] octets in length	0xe33b	32-bits
GET_TX_PKTS_512_1023	Get the total number of packets transmitted (IBP) that are [512 - 1023] octets in length	0xe33c	32-bits
GET_TX_PKTS_1024_1518	Get the total number of packets transmitted (IBP) that are [1024-1518] octets in length	0xe33d	32-bits
GET_TX_PKTS_1519_MAX	Get the total number of packets transmitted (IBP) that are >1518 octets in length	0xe33e	32-bits
GET_TX_DEFERRED_ERR	Get the total number of times; the initial transmission attempt of a frame is postponed due to another frame already being transmitted on the Ethernet network. (HdM)	0xe33f	32-bits
GET_TX_TOTAL_COLLISION	Get the sum of all collision events. (HdM)	0xe340	32-bits
GET_TX_SINGLE_COLLISION	Get the number of successfully transmitted frames, on a particular interface where the transmission is inhibited by exactly one collision (HdM)	0xe341	32-bits
GET_TX_MUL_COLLISION	Get the number of successfully transmitted frames, on a particular interface for which transmission is inhibited by more than one collision. (HdM)	0xe342	32-bits
GET_LATE_COLLISION	Get the number of times, a collision is detected on a particular interface later than 512 bit-times into the transmission of a packet. Such frame are terminated and discarded (HdM)	0xe343	32-bits
GET_TX_EXCV_COLLISION	Get the number of frames, which collides 16 times and is then discarded by the MAC. Not effecting TxMultipleCollisions (HdM)	0xe344	32-bits
GET_TX_EXCV_DEFERRED_ERR	Get the number of times frame, for which transmission is postponed more than 2*MaxFrameSize due to another frame already being transmitted on the Ethernet network. This causes the MAC to discard the frame. (HdM)	0xe345	32-bits
GET_TX_EXCV_LEN_DROP	Get the number of frame, for which transmissions aborted by the MAC because the frame is longer than maximum frame size.	0xe346	32-bits
GET_TX_UNDERRUN	Get the number of internal TX error, which causes the MAC to end the transmission before the end of the frame because the MAC did not get the needed data in time for transmission. The frames are lost and a fragment or a CRC error is transmitted.	0xe347	32-bits

Table 2-21. MAC Transmit Statistics Counters ioctls (Sheet 2 of 2)

IOCTL Commands MAC TX Stat ioctls	Description	Defined Value	Buffer Size
GET_TX_VLAN_TAG	Get the number of OK frames with VLAN tags.	0xe348	32-bits
GET_TX_CRC_ERR	Get the number of frames, which are transmitted with a legal size, but with the wrong CRC field (also called FCS field)	0xe349	32-bits
GET_TX_PAUSE_FRAME	Get the number of Pause frames transmitted.	0xe34a	32-bits
GET_FC_COLLISION_SEND	Get the number of times the collision is generated on purpose on incoming frames, to avoid reception of traffic, while the port is in half-duplex and has flow control enabled, and have not sufficient memory to receive more frames. (HdM)	0xe34b	32-bits

2.7.4.4 Global Status and Configuration ioctls

The following table describes the `ioctl` commands used for configuration and monitoring the port status.

Table 2-22. ioctl Commands for Accessing Global Status and Configuration Registers (Sheet 1 of 2)

IOCTL Commands Global Stat and Config ioctls	Description	Defined Value	Buffer Size
SET_PORT	Set the control register for each port in Vallejo device. To make a port active the bit is set to high. Bit 3:0 [a]	0xe21b	32-bits
GET_PORT	Get the Port status	0xe34c	32-bits
SET_INTERFACE_MODE	Set bit 3:0 of corresponding register for the PHY interface mode. 0 = Fiber, and 1 = Copper	0xe21c	32-bits
GET_INTERFACE_MODE	Get the PHY interface mode for individual port	0xe34d	32-bits
GET_LINK_UP_STATUS	Each bit from 3:0 of the 32-bit corresponding status register records the status of the Link Flag for a given port. This command reads this to get the status of the individual ports. 1 = Link is established	0xe34e	32-bits
SET_RESET_CORE_CLOCK	Activate/inactivate the soft reset for the core clock system.	0xe21d	32-bits
GET_RESET_CORE_CLOCK	Get the status of the soft reset for the core cloak system.	0xe34f	32-bits
SET_PAUSE_BEHAVIOR	Set behavior of the individual port on receiving the Pause Packet. Bit 19:16 Pause Packet Forward Bit 3:0 Pause Packet Corruption	0xe21e	32-bits
GET_PAUSE_BEHAVIOR	Get the Pause packet behavior	0xe350	32-bits
SET_MAC_SOFT_RESET	Activate per port software reset of the MAC core	0xe21f	32-bits
GET_MAC_SOFT_RESET	Get the status of the software reset of the MAC core.	0xe351	32-bits
SET_MDIO_RESET	Activate the software reset of the MDIO module	0xe220	32-bits
GET_MDIO_RESET	Get the status regarding the reset activation of the MDIO module	0xe352	32-bits
SET_UI_ENDIAN_MODE	Set microprocessor Endian. 0 = little Endian, 1 = big Endian	0xe221	32-bits
GET_UI_ENDIAN_MODE	Get microprocessor Endian mode	0xe353	32-bits

Table 2-22. ioctl Commands for Accessing Global Status and Configuration Registers (Sheet 2 of 2)

IOCTL Commands Global Stat and Config ioctls	Description	Defined Value	Buffer Size
SET_LED_MODE	Set the LED mode Bit 1: Enable/Disable LED block Bit 0: LED Control	0xe222	32-bits
GET_LED_MODE	Get LED status	0xe354	32-bits
SET_LED_FLASH_RATE	Set LED flash rate, 00 = 100 ms flash rate 01 = 250 ms flash rate 10 = 500 ms flash rate 11 = Reserved	0xe223	32-bits
GET_LED_FLASH_RATE	Get LED flash rate	0xe355	32-bits
SET_LED_FAULT_ACTION	Set per-port fault disable Disable/enable the LED flashing for local or remote faults	0xe224	32-bits
GET_LED_FAULT_ACTION	Get per-port LED fault status	0xe356	32-bits
GET_JTAG_ID	Get the device identification (fixed here)	0xe357	32-bits

a. Bit position M:N corresponds to the port number, where M = N + 3, with one to one mapping. Means bit N corresponds to port 0, bit N+1 corresponds to port 1, and so on.

2.7.4.5 RX FIFO Configuration ioctl Commands

The following table describes the ioctl commands used to configure the receive FIFO, and to get the status of the receive FIFO.

Table 2-23. ioctl Commands to Configure the RX FIFO (Sheet 1 of 2)

IOCTL Commands RX FIFO Register ioctls	Description	Defined Value	Buffer Size
SET_RFIFO_HIGH_WATERMARK	Set high watermark for RX FIFO. [a]	0xe225	32-bits
GET_RFIFO_HIGH_WATERMARK	Get RX FIFO high watermark level.	0xe358	32-bits
SET_RFIFO_LOW_WATERMARK	Set low watermark for RX FIFO.	0xe226	32-bits
GET_RFIFO_LOW_WATERMARK	Get the RX FIFO low watermark level.	0xe359	32-bits
GET_RX_FRAME_REMOVED	Get the number of frames lost/removed on individual port when RX FIFO on this port becomes full or reset.	0xe35a	32-bits
SET_RX_FIFO_PORT	Set the soft reset register for each port in the RX block. Bit 3:0	0xe227	32-bits
GET_RX_FIFO_PORT	Get the soft reset status in the RX block.	0xe35b	32-bits
SET_RX_FIFO_ERR_FRAME_STT	Set the action to be taken on receiving errored packets, whether such packets are to be dropped or not. Bit 3:0 1 = Frame Drop Enable 0 = Frame Drop Disable	0xe228	32-bits
GET_RX_FIFO_ERR_FRAME_STT	Get the status of the action to be specified on receiving the errored packets.	0xe35c	32-bits

Table 2-23. ioctl Commands to Configure the RX FIFO (Sheet 2 of 2)

IOCTL Commands RX FIFO Register ioctls	Description	Defined Value	Buffer Size
GET_RX_FIFO_OVERFLOW_STT	Get the RX FIFO status, if a FIFO full situation has occurred. The corresponding register is cleared on read. Bit 3:0	0xe35d	32-bits
GET_OUT_SEQUENCE_INFO	Get the status of the RX FIFO, when out of sequence data is detected in the RX FIFO. The corresponding register is cleared on read. Bit 3:0	0xe35e	32-bits
GET_DROPPED_PKTS	Get the number of packets dropped by the RX FIFO due to various errors.	0xe35f	32-bits
GET_RW_PTR_RX_FIFO	Get the value for the read and write pointer for the RX FIFO.	0xe360	32-bits
GET_OCCUPANCY_RX_FIFO	Get the occupancy for RX FIFO. The corresponding register is read only.	0xe361	32-bits
GET_CAPTURED_PKT_LEN	Get the length information of the captured packet (in bytes) at four ports. The byte position equals to the port number.	0xe362	32-bits
GET_INDIRECT_ADR_CTL	The corresponding register provides the indirect memory access for CPU to read captured data.	0xe363	32-bits
GET_READ_DATA	Get 8 bytes of the read data.	0xe364	64-bits
SET_CAPTURE_ENABLE_RX_FIFO	Set the capture and loop back feature at different ports. Bit 11:8 = Loopback enable. Bit 7:0 = Capture Enable Mode, each pair of bit corresponds to port number from LSB.	0xe229	32-bits
GET_CAPTURE_ENABLE_RX_FIFO	Get the status of the capture enable and loopback feature.	0xe365	32-bits
SET_PRE_PENDING_CRC_ENABLE	Set the corresponding register to prepend every packet with two extra bytes and also enable the CRC stripping of the packets. Bit 7:4 = Enable CRC stripping. Bit 3:0 = Enable pre-pending, Pre-pending should not be enabled in loopback mode.	0xe22a	32-bits
GET_PRE_PENDING_CRC_ENABLE	Get the status of the pre-pending and CRC stripping feature.	0xe366	32-bits
SET_MATCHING_PATTERN	Set the matching pattern, which is checked with the TYPE/LEN fields of every incoming packet to capture specific packets from data traffic.	0xe22b	32-bits
GET_MATCHING_PATTERN	Get matching pattern, wet by the previous ioctl command.	0xe367	32-bits
SET_JUMBO_PKT_SIZE	Set the jumbo packet size in 8 byte location.	0xe22c	32-bits
GET_JUMBO_PKT_SIZE	Get the jumbo packet size set by the previous ioctl command.	0xe368	32-bits
GET_PKT_DROP_CAP_FIFO	Get the number of packets dropped at capture FIFO due to FIFO full or bad packets or during CPU not read the previous captured packet.	0xe369	32-bits

a. The Port Number should be given separately, as a parameter to the buffer, whose pointer is passed as an argument to the ioctl command.

2.7.4.6 TX FIFO Configuration Ioctls

The following table describes the ioctl commands used to configure and monitor the transmit FIFO.

Table 2-24. IOCTL List to Configure and Monitor the TX FIFO (Sheet 1 of 2)

IOCTL Command TX FIFO Register ioctls	Description	Defined Value	Buffer Size
SET_TFIFO_HIGH_WATERMARK	Set high watermark for TX FIFO, for each port separately.	0xe22d	32-bits
GET_TFIFO_HIGH_WATERMARK	Get high watermark for TX FIFO	0xe36a	32-bits
SET_TFIFO_LOW_WATERMARK	Set low watermark for TX FIFO, for each port separately.	0xe22e	32-bits
GET_TFIFO_LOW_WATERMARK	Get low watermark for TX FIFO.	0xe36b	32-bits
SET_MAC_THRESHOLD	Set the MAC threshold for TX FIFO.	0xe22f	32-bits
GET_MAC_THRESHOLD	Get the MAC threshold TX FIFO value.	0xe36c	32-bits
GET_TX_FIFO_OVERFLOW_STT	Get the status information as Bit 11:8 FIFO out of sequence event trace record Bit 7:4 FIFO underflow event trace record Bit 3:0 FIFO Overflow event trace record.	0xe36d	32-bits
SET_LOOP_RX_TX	Set the respective bit high to perform the external loopback. Bit 3:0 0 = Normal Operation 1 = The SPI-3 data coming from the RX block is sent to the TX FIFO instead of the SPI-3 Receive interface	0xe230	32-bits
GET_LOOP_RX_TX_STT	Get external loopback status	0xe36e	32-bits
SET_TX_FIFO_PORT	Assert/De-assert reset for each port in TX block. Bit 3:0 set to low to make port active.	0xe231	32-bits
GET_TX_TFIFO_PORT	Get status of the port	0xe36f	32-bits
GET_TX_DROP_FRAME	Get the number of frames lost/removed, when TX FIFO on individual port becomes full or reset. This register is clear on read.	0xe370	32-bits
GET_TX_DROP_PKTS	Get the number of packets dropped by the TX FIFO of individual port, due to various errors. This register is cleared on Read.	0xe371	32-bits
GET_TX_RW_PTR	Get the value of the read write pointer for the TX FIFO of individual port. This register is cleared on read.	0xe372	32-bits
GET_TX_OCCUPANCY	Get the occupancy for the TX FIFO . The corresponding register is read only.	0xe373	32-bits
SET_TXINSERT_DATA	Insert the 8 bytes data for port 0	0xe232	64-bits
GET_TXINSERT_DATA	Get the inserted 8-bytes data for each port separately.	0xe374	64-bits
SET_TXFIFO_INFO_ADR	Set the indirect memory access for CPU to write/read data to/from individual insertion FIFO port. Bit 10 = Reset Bit 9 = Write Bit 8 = Read Bit 7:3 = Address Bit 2:0 = Info	0xe233	32-bits
GET_TXFIFO_INFO_ADR	Get the above defined status	0xe375	32-bits

Table 2-24. IOCTL List to Configure and Monitor the TX FIFO (Sheet 2 of 2)

IOCTL Command TX FIFO Register ioctls	Description	Defined Value	Buffer Size
SET_TXFIFO_DROP_INSERT	Enable independently, the individual TX FIFO to drop the erroneous packet and insertion of packet through insertion FIFO. Bit 7:4 = Set high to enable read from insertion FIFO. Bit 3:0 = Set high to discard the error packets in TX FIFO.	0xe234	32-bits
GET_TXFIFO_DROP_INSERT	Get the above defined feature in corresponding SET ioctl.	0xe376	32-bits
SET_TX_MINI_FRAME_SIZE	Set the different minimum length of the packets to be transmitted to MAC independently. These values are used to pad short packets if padding is enabled. Bit 19:16 = Set bit high to enable padding of short packets. Bit 15:12 = (for port 3) If the programmed value is ‘N’ then the minimum number of bytes in packet is equal to ‘N * 8’ bytes. Where N = A, B, C, D and E Same as above, bit 11:8,7:4 and 3:0 are for port 2,1,0 respectively.	0xe235	32-bits
GET_TX_MINI_FRAME_SIZE	Get the minimum length of the packet to be transmitted to the MAC.	0xe377	32-bits

2.7.4.7 MDIO Interface Related ioctl Commands

The following table describes the ioctl commands to configure and monitor the MDIO interface.

Table 2-25. IOCTLs to Configure and Monitor MDIO Interface

IOCTL Commands MDIO Interface Ioctls	Description	Defined Value	Buffer Size
SET_MDIO_CMD_ADDR	Bit 20 = Set high to perform operation Bit 17:16 = Identify operation to be performed. Bit 9:8 = address of external device Bit 4:0 = Reg Address	0xe236	32-bits
GET_MDIO_CMD_ADDR	Get that value of the MDIO command and address register.	0xe378	32-bits
SET_MDIO_SINGLE_RW_DATA	Bit 31-16 = MDI Read data from external device Bit 15:0 = MDI write data to external device	0xe237	32-bits
GET_MDIO_SINGLE_RW_DATA	Get MDI read write data	0xe379	32-bits
SET_AN_PHY_ADDR	Set the PHY address enable Bit 3:0 = set high to enable PHY address []	0xe238	32-bits
GET_AN_PHY_ADDR	Get the PHY address status	0xe37a	32-bits
SET_MDIO_CTL	Bit 19:16 = Remote Fault Status Bit 3 = MDI Progress Bit 2 = Set high to enable MDI Bit 1 = set high to enable auto-scan Bit 0 = select speed of MDC clock	0xe239	32-bits
GET_MDIO_CTL	Get the MDIO Control status	0xe37c	32-bits

2.7.4.8 SPI-3 Configuration ioctl Commands

The following tables describes the `ioctl` commands to configure and monitor SPI-3 interface.

Table 2-26. List of `iotcl` Commands to Configure the SPI-3 Interface

IOCTL Commands SPI-3 Configure Ioctls	Description	Defined Value	Buffer size
SET_SPI3_TX_GLOBAL_CONFIG	Set the SPI3 Transmitter and Global configuration (4x8 mode)	0xe23a	32-bits
GET_SPI3_TX_GLOBAL_CONFIG	Get the SPI3 Transmitter and Global configuration (4x8 mode)	0xe37d	32-bits
SET_SPI3_RX_CONFIG	Configure the SPI-3 Receiver	0xe23b	32-bits
GET_SPI3_RX_CONFIG	Get the SPI-3 Receiver configuration	0xe37e	32-bits
GET_SPI3_TX_INT_STATUS	Get the status of various SPI-3 transmit error interrupts. (one for each port. []	0xe37f	32-bits
SET_SPI3_TX_INT_ENABLE	Configure the interrupt enable for the various interrupt states.[]	0xe23c	32-bits
GET_SPI3_TX_INT_ENABLE	Get the interrupt status,	0xe380	32-bits
GET_SPI3_ADR_PARITY_ERROR	Get the number of packets dropped sue to address parity error.	0xe381	32-bits
GET_SPI3_PKT_DISABLE_PORT	Get number of packets received for disabled port that has been dropped. []	0xe382	32-bits
GET_SPI3_PKT_SYNC_ERR	Get the number of packets received with full SYNC error (No SOP but EOP) that has been dropped.	0xe383	32-bits
GET_SPI3_PKT_SHORT_DROP	Get the number of dropped, whose length is less than 9 bytes.	0xe384	32-bytes

2.7.4.9 SERDES Interface ioctls

The following table describes the ioctl commands to configure and monitor SerDes interface.

Table 2-27. IOCTLs used to configure SerDes Interface (Sheet 1 of 2)

IOCTL Commands SERDES Interface Ioctls	Description	Defined Value	Buffer Size
SET_ACDC_COUPLING	Set AC or DC coupling on the output of each SerDes port (Tx and RX are independent) Bit 7:0 = each pair of bits represents the port number from LSB, and out of that even bit number is for TX and odd is for RX	0xe23d	32-bits
GET_ACDC_COUPLING	Get the AC or DC coupling status	0xe385	32-bits
SET_SERDES_TX_DRV_COEFF	Set the various programmable strength s on each of the SerDes port	0xe23e	32-bits
GET_SERDES_TX_DRV_COEFF	Get the strength on each of the SerDes Port	0xe386	32-bits
SET_TX_DRV_POW_LEVEL	Set the power level for each of the SerDes port. Each byte corresponds to the port number starting from LSB.	0xe23f	32-bits
GET_TX_DRV_POW_LEVEL	Get the power level for each port	0xe387	32-bits
SET_TX_LINK_VALIDATION	Configure the link status, and stores that.	0xe240	32-bits
GET_TX_LINK_VALIDATION	Get the status of the link validation	0xe388	32-bits

Table 2-27. IOCTLs used to configure SerDes Interface (Sheet 2 of 2)

IOCTL Commands SERDES Interface Ioctls	Description	Defined Value	Buffer Size
SET_TX_RX_POW_DOWN	Set the Power-down TX and RX power-down bits to allow per port power-down of the unused port. Bit 13:10 = Set bit to high to Tx Power down per port (each bit from LSB corresponds to each port number.) Bit 3:0 = set bit high to RX Power-down per port, port mapping to the bit is same as above.	0xe241	32-bits
GET_TX_RX_POW_DOWN_STT	Gets status of each port regarding individuals TX and RX power down.	0xe389	32-bits
SET_RX_DATA_SYNC	Enable the feature, which allows the incoming data stream to be slipped by one bit each time the signal is activated. Bit 3:0 = Set bit high to activate the data synchronization control per port. Each bit from LSB maps to the port number.	0xe242	32-bits
GET_RX_DATA_SYNC	Get the receive data synchronization status.	0xe38a	32-bits
SET_RX_LINK_VALIDATION	Setting these bits allows a BIST test to be carried out to validate the link function. Bit 23:20 = Set high to enable link validation. Each bit from LSB maps to the respective port number. Bit 13:10 = Set high to reset link validation controller. Each bit from LSB maps to the respective port number. Bit 3:0 = set high to enable link validation loopback.	0xe243	32-bits
GET_RX_LINK_VALIDATION	Get the receive validation status.	0xe38b	32-bits
GET_RX_LINK_STT	Same as above, but corresponding register is not cleared when read.	0xe38c	32-bits
SET_RX_PHASE_ROT	Control the Phase Rotator in the SerDes Rx on a per port basis. Bit 23:20 = Set high to enable phase rotator retard. Each bit from LSB maps to the respective port number Bit 13:10 = Set high to enable phase rotator. Each bit from LSB maps to the respective port number. Bit 3:0 = set high to enable phase rotator advance. Each bit from LSB maps to the respective port number	0xe244	32-bits
GET_RX_PHASE_ROT	Get the phase rotator status.	0xe38d	32-bits
GET_RX_PHASE_ROT_BUS	Get phase rotator state in conjunction with RX phase rotator control.	0xe38e	32-bits
SET_RX_LATCH_OBSRV_01	Allow the capture of data at the output of the de-serializer SerDes for port 0 and 1.	0xe245	32-bits
GET_RX_LATCH_OBSRV_01	Get the latch observation for the port 0 and 1.	0xe38f	32-bits
SET_RX_LATCH_OBSRV_23	Allow the capture of data at the output of the de-serializer SerDes for port 2 and 3.	0xe246	32-bits
GET_RX_LATCH_OBSRV_23	Get the latch observation for the port 2 and 3.	0xe390	32-bits
GET_RX_SIGNAL_LEVEL	Get the status of the Rx input in relation to the level of the signal being received from the line. Bit 3:0 = High bit status depicts Signal, while low for Noise. . Each bit from LSB maps to the respective port number	0xe391	32-bits
GET_SERDES_TX_CONFIG	Get the default TX block configuration value.	0xe392	32-bits
GET_SERDES_RX_CONFIG	Get the default RX block configuration value	0xe393	32-bits
GET_PLL_LOCK	Get the status of the PLL lock for the RX and TX block.	0xe394	32-bits

2.7.4.10 GBIC Module Interface ioctls

The following table describes the ioctl commands to control and monitor the interface to the GBIC modules when used in SerDes mode.

Table 2-28. IOCTLs to control and monitor GBIC module

IOCTL Command GBIC Interface Ioctls	Description	Defined Value	Buffer Size
GET_GBIC_STAUS	Get the interface status to the GBIC module when used in SerDes mode.	0xe395	32-bits
SET_GBIC_CTL	Configure the GBIC module	0xe247	32-bits
GET_GBIC_CTL	Get the GBIC module configuration	0xe396	32-bits
SET_I2C_CTL_DATA,	Set the I2C control data	0xe248	32-bits
GET_I2C_CTL_DATA,	Get the I2C control Data	0xe397	32-bits

2.8 Support for Multiple Quad Gigabit Ethernet I/O Cards

The driver supports the existence of the multiple Quad Gigabit Ethernet I/O cards on the baseboard. These two cards can sit on the Media interface, and on the Switch Fabric interface. Both the cards are identical. The individual card information is opaque from the application, since the application treats both cards as merged and interprets that there are a total of eight ports supported. The application passes the port number from [0-7] to access the ports. This port number is parsed in the driver, and is identified for the respective card. Card 0 implies the Quad Gigabit Ethernet I/O card on the Media interface, and the Card 1 implies the Quad Gigabit Ethernet I/O card on the Switch Fabric interface.

Table 2-29. Individual Port Information Interpretation

Port Number passed by the Application	Associated PtLone media card	Port number on the respective Media Card
0-3	Card 0	0-3
4-7	Card 1	0-3

In this kernel mode, the driver treats both cards as a single device. The differentiation is made on the minor number assigned to the ports. For the minor number [0-3], card 0, (the Quad Gigabit Ethernet I/O card) on the Media interface is referenced, and for the minor number [4-7], card 1, which is on the Switch Fabric interface, is referenced.

2.9 System Dependencies & File Structures

The main system files used are as follows:

- module.h – used for dynamic loading of the modules into kernel
- kernel.h – contains the function prototype
- fs.h – defines the file table structures

The implementation of the driver as a loadable module requires the following operating system primitives for registering and removing the device.

2.9.1 Device Register Routine

The following routine is called in the `Init_Module()` function internally, while the *`insmod`* command is executed.

```
int register_chrdev(uint32 major, const char *name, struct
file_operations *fops);
```

This routine registers the device under a free major number, as returned by the kernel. The "major" argument is the major number being requested, "name" is the name of the device, which appears in /proc/devices, and "fops" is the pointer to an array of function pointers, used to invoke driver's entry points.

2.9.2 Device Unregister Routine

When a module is unloaded from the system, the major number must be released. This routine is used to unregister the device.

```
int unregister_chrdev(uint32 major, const char *name);
```

This routine is called from the module's cleanup function. The argument "major" is the number being released and the "name" is the name of the associated device.

2.9.3 Interrupt Handling Routine

In the kernel mode driver, the ISR are implemented using the system functions. This ISR generates a signal, "SIGIO" on getting an interrupt, and this signal is handled in the application that uses the driver.

2.10 Exported Kernel APIs

This section defines the ways in which external APIs interface with the module being designed. Internal function routines are the same and applicable for both the modes. The basic difference lies in their calling functions, and the parameters passed to the respective APIs.

The following table describes the APIs and their respective brief description for the driver in both the modes. It gives an overview of the common functionality between the kernel mode and user mode driver.

Table 2-30. APIs provided by the driver in Kernel mode

API Name	Description
`Init Module`	This function is invoked on inserting the driver module (using insmod). This routine registers the driver as a character device, initialize configure the Vallejo MAC's, and enable all ports
`GbEMAC_open`	This routine is invoked on the device open call. This routine updates the status of the port to open and activated. It registers the ISR to the interrupt vector.
`GbEMAC_ioctl`	This function is the entry point to perform configuration and get status of the device.
`GbEMAC_close`	This routine is invoked on the device close call. This routine sets the status of the device to deactivate and the port is no longer being in normal operation. It unregistered the ISR from the interrupt vector.
`GbEMAC_i2s_fasync`	This function is called by the system call "fcntl" to register the user application with the specific device to receive the signal, if the interrupt comes from this device.
`Cleanup Module`	This function is invoked when the module has to be removed from the kernel. It un-registers the device and de-allocates memory. Before closing the device, it checks if the ISR is still connected to interrupt vector, if so, it un-registers the ISR before unregistering the driver.

2.10.1 Init Module

This function is invoked on inserting the driver module (via *insmod*).

Syntax

```
uint32 init_module (void)
{
return gbe_mac_error;
}
```

Input

NULL

Returns

Return Type	• gbe_mac_error Code

Example

```
uint32 init_module (void)
{
Checks the presence of the device by reading I2C EEPROM.
Registers the driver as a character device with the kernel.
Verifies the slow port access to the registers of the MAC.
Connects the ISR to the interrupt vector.
Calls gbe_mac_config (device_number, port_mask, port_mode) routine to
configure:
```

```
Top level global registers.
Exits giving gbe_mac_error, if above calls fails.
Configure  the requisite per port MAC registers.
Assign the MAC address to each port.
Call Delay ()to Introduce 1 µSec delay for clock to stabilize.
Make port start by enabling them.
Exits giving gbe_mac_error, if above calls fails anywhere.
Exits giving ERROR CODE if above calls fails.
Performs error check at each step above mentioned, and generates ERROR CODE on
error.

return gbe_mac_error;
}
```

2.10.2 GbEMAC_open

This function is invoked on the device open call. It checks the port status before opening the port. The very first call to this function registers the interrupt handler routine to the in the kernel.

Syntax

`uint32 GbEMAC_open (struct inode *GbEMAC_inode,struct file *GbEMAC_file)`

Inputs

`struct inode`	This pointer points to the inode structure defined in the linux/fs.h system header file. It includes the mount structures. This is used to retrieve the minor number.
`struct file`	This pointer points to a file structure, which defines the set of functions implemented in this driver. Here `GbEMAC_open`, `GbEMAC_close`, and `GbEMAC_ioctls` are the elements of this structure, since these functions are implemented.

Returns

Return Type	• gbe_mac_error Code • Success

Example

```
uint32 GbEMAC_open (struct inode *GbEMAC_inode,

{

struct file *GbEMAC_file)
Increments the usage count.
Retrieves the minor number from the GbEMAC_inode.
Calculates the card for which the open call is intended.
Checks the device state, and if a new port has to open, calls the associated
ioctl to enable the particular port.
Checks error condition at each step, and generates ERROR CODE if any error has
occurred.
```

```
return SUCCESS;
}
```

2.10.3 GbEMAC_close

This function is invoked on the device close call. This routine will set the status of the device to deactivate and the port will no longer be in normal operation.

Syntax

```
uint32 GbEMAC_close (struct inode *GbEMAC_inode,
                                        struct file *GbEMAC_file)
```

Input

Struct inode*	This pointer points to the inode structure defined in the linux/fs.h system header file. It includes the mount structures. This is used to retrieve the minor number.
Struct file*	This pointer points to a file structure, which defines the set of functions implemented in this driver. Here GbEMAC_open, GbEMAC_close, and GbEMAC_ioctls are the elements of this structure, since these functions are implemented.

Returns

Return Type	gbe_mac_error Code

Example

```
uint32 GbEMAC_close (
                                        struct inode *GbEMAC_inode,
                                        struct file *GbEMAC_file)
{
Decrement the usage count
Retrieve the minor number from the GbEMAC_inode
Set the port as CLOSED, which are not used by any other application.
Check error condition at each step, and generate ERROR CODE if any error has
occurred.
return SUCCESS;
}
```

2.10.4 GbEMAC_ioctl

This routine is provided to call an `ioctl`. This routine calls the ***config_handler*** routine for the implementation of the `ioctl` command.

Syntax

```
uint32 GbEMAC_ioctl (struct inode *GbEMAC_inode,
                                   struct file * GbEMAC_file,
                                   uint32 ioctl_command,
                                   void *ioctl_struct);
```

Input

struct inode * *GbEMAC_inode*	Pointer to inode structure
struct file * *GbEMAC_file*	Pointer to file structure
uint32 *ioctl_command*	The ioctl number to be implemented
void * *ioctl_struct*	Points to the IOCTL_PTR, which is typecast to the void pointer

Returns

Return Type	• gbe_mac_error Code

Example

```
uint32 GbEMAC_Ioctl (
                                    struct inode *GbEMAC_inode,
                                    struct file * GbEMAC_file,
                                    uint32 ioctl_command,
                                    void * ioctl_struct)
{
Check for the port state to which this ioctl intended,
If port close, log error message,
Call gbe_mac_config_handler (arg_ioctlCommand, arg_pIoctlStruct); for actual
implementation of the ioctl command
Check error condition at each step, and generate ERROR CODE if encounters any
error
        return gbe_mac_error;
}
```

2.10.5 GbEMAC_i2s_fasync

This function is called by the system call “fcntl” to register the user application with the specific device to receive the signal, if the interrupt comes from this device. This function internally stores the “fd” of the calling application in a queue. The driver sends the signal SIGIO to the user processes that have called this function with ASYNCH flag.

Note: This function is called by the application for receiving notification through a signal SIGIO of the occurrence of interrupt regarding change in link status. The application callback handler should call the IOCTL GET_LINK_UP_STATUS to get port status.

2.10.6 Cleanup Module

This function is invoked when the module is removed from the kernel. It will unregister the device and deallocate memory.

Syntax

```
void cleanup_module (void)
```

Input

NULL

Return

NULL

Example

```
void cleanup_module (void)
{
First check the usage count; kernel will never be able to unload the module if
the counter doesn't drop to zero.
Unregister the driver from the kernel, iff usage count is zero.
Check for the Interrupt handler, if that is still installed, then uninstall
that Interrupt Handler.
Check error code at each step, and generate ERROR CODE on encountering any
error
return NULL;
}
```

2.11 Interrupt Handling

The Linux kernel provides the routine ***`request_irq( ),`*** which connects a routine to a hardware interrupt. Its counter part is ***`free_irq ( )`*** which frees the connected interrupt. It connects a user defined C routine to an interrupt vector. These system routines are provided in `<linux/sched.h>` header file. A routine connected to an interrupt is called an Interrupt Service Routine (ISR).

This ***`request_irq ( )`*** routine is called in the ***`GbEMAC_open( )`*** when the device is first opened, ***`init_module ( ).`*** When an interrupt is generated, it is captured by the system, and intern it set the corresponding bit high in the interrupt register. This intern calls the ***`gbe_mac_isr ()`*** function, which is connected to the interrupt vector. This function first detects that which interrupt is generated and is intended for which gigabit MAC port, and then calls the related ISR for the appropriate action. Here, the corresponding function conveys the message related to the interrupt. The ***free_irq ( )*** is called just before closing the device in the ***cleanup_module ( )***.

2.12 Functions to Access the Kernel Mode Driver

The following system calls are used to access kernel driver APIs from mode applications. These functions expect a character device to be opened using the `open()` function. Applications can then send different commands to get or set values in the kernel driver. The system call `close()` is used to close the device.

2.12.1 Open( )

This function is called by the application to open the device, which in turn calls the `GbEMAC_open()` API to open the device. See Section

Syntax

```
int open( const char *pathname, int flags);
```

Input

`pathname`	The name of the character device file to be opened.
`flags`	Mode in which the character device file to be opened, [O_RDWR].

2.12.2 Close( )

This function is called by the application to close the device, which in turn calls `GbEMAC_close()` API to close the device. See Section

Syntax

```
int close( int fd);
```

Input

`fd`	The file descriptor returned by the open call earlier.

2.12.3 ioctl( )

This function is called by the application to call the ioctl command, which in turn Calls `GbEMAC_ioctl()` to send set/get commands. See `GbEMAC_ioctl`.

Syntax

```
int ioctl( int fd, unsigned int IoctlCmd, void * IoctlPtr);
```

Input

`fd` The file descriptor returned by the open call earlier.

`IoctlCmd` The IOCTL command to be executed.

`IoctlPtr` The pointer of the user defined structure is type caste'd and passed to the kernel using this command. This structure is updated by the kernel mode driver with the appropriate result value.

2.12.4 Fcntl( )

This function registers the user callback functions to an array, and when the interrupt occurs, all the respective functions are called. Internally this call calls `GbEMAC_i2s_fasync()` API for callback registration. See Section

Syntax

```
int fcntl( int fd, int cmd, long arg);
```

Input

`fd` The file descriptor returned by the open call earlier.

`cmd` Command to be performed.

`arg` Passed the PID of itself, getting by the getpid() function call.

10-Port Gigabit Ethernet Media Card 3

The IXD2810 is a 10-port Gigabit Ethernet (GbE) add-on media card for the IXDP2800 Network Processor advanced development platform. The IXD2810 consists of a media interface to the network processors of the IXDP2800, an IXF1110 MAC and line interfaces that connect to transmit (Tx) and receive (Rx) optical fibers.

The scope of the device driver is limited to initialization and configuration of the MAC device on the IXD2810 card and providing an I/O control path to access IXD2810 registers. The driver does not support any data-path functionality.

Refer to the following sections for operating system-specific IXD2810 device driver API information:

- Section 3.1, “Linux Environment”
- Section 3.2, “VxWorks Environment”

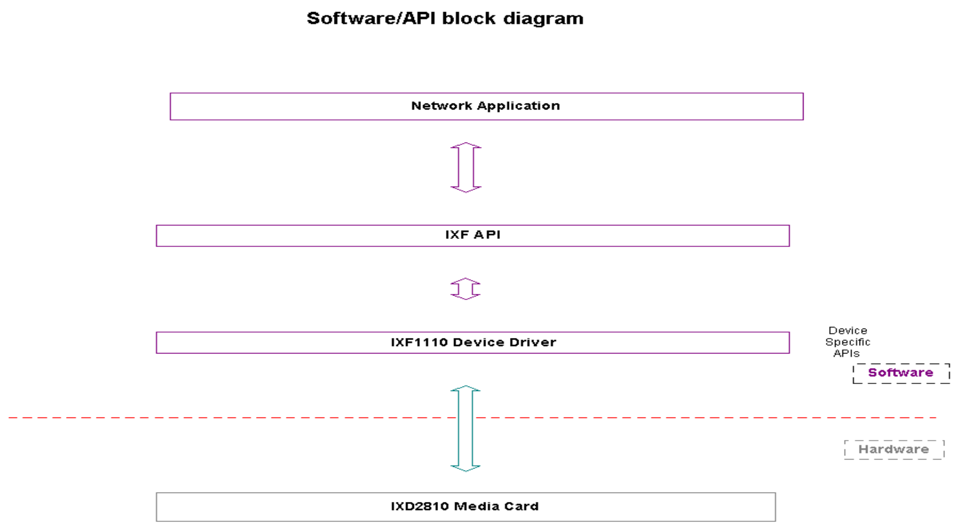
3.1 Linux Environment

The IXD2810 device driver discussed in this section is for the Monta Vista* Linux operating system on the Intel® IXDP2800 Advanced Development Platform.

3.1.1 Design Decomposition

The IXD2810 device driver and the supporting software is designed to be modular and portable. There is no need for synchronization between the Master and Slave NPUs.

Figure 3-1. Device Driver Design



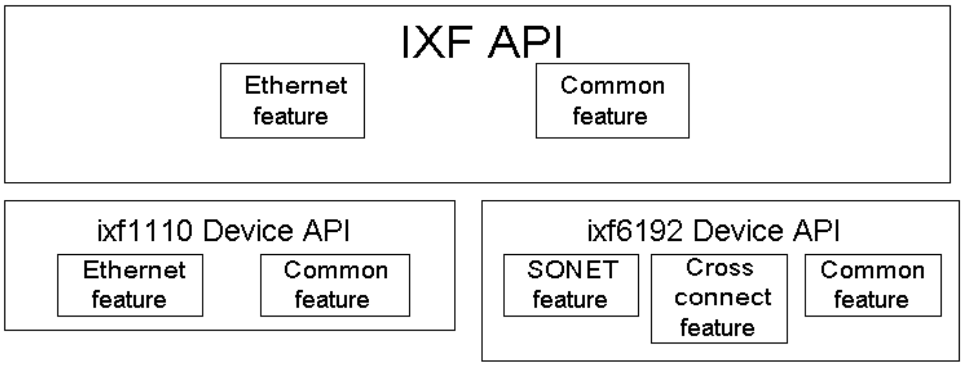
The device driver for the IXD2810 has the following components:

- IXF API module
- IXF1110 device-specific driver
- IXF1110 interrupt-specific module

3.1.2 IXF API Module

The IXF API provides a common and consistent interface for supported IXF devices. For each supported device, the IXF API provides device specific functions, as well as common interface functions for features that are not specific to a particular device. The following diagram illustrates the IXF API, feature API, and device API layers.

Figure 3-2. IXF API Model



The IXF API provides IXF1110-specific functions for general device configuration, while access to the Ethernet functional block of the IXF1110 is provided by interface functions that are coupled with the functionality of the block in question. In addition, the IXF API also provides generic read and write access to the devices.

The IXF API interfaces to multiple devices (whether of the same or different types) simultaneously. The IXF API provides the application with a 'chip ID' that is used to reference the device it is accessing.

The IXF API itself provides little implementation of any functionality. Instead, it directs the call to the appropriate device API, which in turn implements the IXF API function for a specific device.

3.1.2.1 Feature APIs

The feature APIs are the upper layers of Figure 3-2. Each 'feature' has its own API, and provides a common interface to the functionality of different devices. Each device may have its own implementation of a feature API, as the implementation may differ from device to device.

Each feature API function signature must match the corresponding IXF API function exactly, as it is a feature API function that actually provides the implementation for the IXF API function. As a result, the application has a common interface to functionality that is shared by several devices. Feature APIs are independent of each other; as for some devices (IXF11100, for example) subsets of the complete feature set must work properly.

There are different types of Feature APIs:·

Table 3-1. Feature API Types

Type	Description
Device specific	Specific to a device. Examples of this include global registers and global configuration.
Common	Common to all (or at least the vast majority of) devices. Functionality includes resetting the device, getting the device ID/version, and generic read and write access.
Functionality based	Provides interfaces for the following blocks: - SPI4 - Ethernet

3.1.2.2 Device APIs

The device APIs are the bottom layers of Figure 3-2. Each device has its own API that provides device-specific implementation of IXF API functions. The Device APIs could be described as 'composite' APIs, as each Device API consists of device- specific functions, plus the feature APIs for all features that the device supports. The Device API function signatures must match the corresponding IXF API functions exactly, as it is a device API function that actually provides the implementation for almost all IXF API functions.

The IXF1110 API module provides API calls to access the chip. This API includes the functions described in the following subsections:

ixf1110Reset

Resets the chip, then reconfigures it.

Syntax

```
extern bb_Error_e
Ixf1110Reset(bb_ChipData_t *pChipData,
                 bb_ChipSegment_t *ptSegment,
                 bb_SelResetType_e resetType);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
ptSegment	bb_ChipSegment_t*	Chip section or block to reset, inclusive of channel (where required)
ResetType	bb_SelResetType_e	Type of reset to perform: bb_RESET_RX_FIFO bb_RESET_TX_FIFO bb_RESET_XGMAC bb_RESET_CORE_CLK bb_RESET_XGMAC_ALL bb_RESET_RX_FIFO_ALL bb_RESET_TX_FIFO_ALL

Returns

bb_Error_e	Error
b_NO_ERROR	Success

ixf1110InitChip

Initializes the chip based upon the configuration passed in by pChipData.

Note: The chip will be set offline (tri-stated and interrupts disabled) while initializing.

Syntax

```
extern bb_Error_e
Ixf1110InitChip(bb_ChipData_t *pChipData,
                 InitRegTable_t *pTable);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
pTable	InitRegTable_t*	Initializes data to be committed to the chip

Returns

bb_Error_e	Error
b_NO_ERROR	Success

ixf1110GetChipInfo

Gets the chip version and ID numbers.

Syntax

```
extern bb_Error_e
Ixf1110GetChipInfo (bb_ChipData_t *pChipData,
                    bb_ChipInfo_t *pChipInfo);
```

Input

pChipData	bb_ChipData_t*	Initializes chip data

Returns

bb_Error_e	Error
b_NO_ERROR	Success
pChipInfo ixf18110_ChipInfo_t*	place to return chip information: IXD2810 board version IXD2810 board ID

ixf1110InitAlarmCallback

Sets the pointer to the Alarm Callback Method. This is a user-defined function that can be called at the end of the `ixf1110_ChipIsr` routine allowing further processing of the collected alarm data.

Syntax

```
extern bb_Error_e
Ixf181110InitAlarmCallback(bb_ChipData_t *pChipData,
                           AlarmCallBack pAlarmCallbackArg);
```

Input

pChipData	bb_ChipData_t*	Initializes chip data
pAlarmCallback	AlarmCallBack	Points to an Alarm Callback function

Returns

bb_Error_e	Error
b_NO_ERROR	Success

ixf1110SetAlarmCfg

Sets the alarm configuration.

Syntax

```
extern bb_Error_e
Ixf1110SetAlarmCfg(bb_ChipData_t *pChipData,
                     bb_ChipSegment_t *section,
                     bb_AlarmType_e AlarmType,
                     void *pAlarmCfg);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
section	bb_ChipSegment_t*	Chip section or block
pAlarmCfg	bb_AlarmType_e	Alarm type to configure
pAlarmCfg	void*	Alarm configuration data

Returns

bb_Error_e	Error
b_NO_ERROR	Success

ixf1110ChipIsr

This function is called to handle all interrupts that have been indicated by the chip. The interrupts are handled according to the hierarchy.

Syntax

```
extern bb_Error_e
Ixf1110ChipIsr(bb_ChipData_t *pChipData);
```

Input

pChipData	bb_ChipData_t*	Initializes chip data

Returns

bb_Error_e	Error
b_NO_ERROR	Success

ixf1110SetCfg

This function sets the configuration of POS watermarks, POS flow control, and the chip's GFC role.

Syntax

```
extern bb_Error_e
Ixf1110SetCfg(bb_ChipData_t *ptChipData,
                 bb_ChipSegment_t *ptSegment,
                 bb_SelConfig_e  SelCfg);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
ptSegment	bb_ChipSegment_t*	Block or section to configure
SelCfg	bb_SelConfig_e	Selects configuration type

Returns

bb_Error_e	Error
b_NO_ERROR	Success

ixf1110GetCfg

This function retrieves configuration information.

Syntax

```
extern bb_Error_e
Ixf1110GetCfg(bb_ChipData_t *ptChipData,
                 bb_ChipSegment_t *ptSegment,
                 bb_SelConfig_e SelCfg);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
ptSegment	bb_ChipSegment_t*	Block or section to retrieve from
SelCfg	bb_SelConfig_e	Status type

Returns

bb_Error_e	Error
b_NO_ERROR	Success

ixf1110GetStatus

Retrieves the status of the Rx AIS, Input Clock Activity, and far-end GFC role.

Note: The test for the far-end GFC Role depends upon having active ATM traffic.

Syntax

```
extern bb_Error_e
Ixf1110GetStatus(bb_ChipData_t *pChipData,
                 bb_ChipSegment_t *section,
                 bb_SelStatus_e selStatus,
                 void *pStatus);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
section	bb_ChipSegment_t*	Block or section to retrieve status from
SelStatus	bb_SelStatus_e	The status type to retrieve

Returns

bb_Error_e	Error
b_NO_ERROR	Success
pStatusvoid*	Place to put Status

ixf1110GetCounters

Retrieves a set of Rx/Tx counters for a selected OHT type, ATM or POS, on a per-channel basis.

Syntax

```
extern bb_Error_e
Ixf1110GetCounters(bb_ChipData_t *ptChipData,
                   bb_ChipSegment_t *ptSection,
                   bb_SelCounters_e eCounter,
                   void *pCounters);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
ptSection	bb_ChipSegment_t*	Chip block or segment
eCounter	bb_SelCounters_e	The set of counters to retrieve

Returns

bb_Error_e	Error
b_NO_ERROR	Success
pCounters void*	A pointer to a Counter Structure that corresponds to the Selected Counters to retrieve.

ixf1110Read

Syntax

```
extern bb_Error_e
Ixf1110Read(bb_ChipData_t *pChipData,
             bb_Word_Size_t wordSize,
             ulong address,
             ushort length,
             void *buffer);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
wordSize	bb_Word_Size_t*	Enum size of data to be read: ONE_BYTE = 1 TWO_BYTES = 2 FOUR_BYTES = 4 EIGHT_BYTES = 8
address	ulong	Offset from chip base address to begin read
length	ushort	Number of words to read
buffer	Void*	Pointer to a structure in which to place the read results

Returns

bb_Error_e	Error
b_NO_ERROR	Success
buffer void*	Buffer contains read results

ixf1110SetChipOnline

Sets one or more of the ports online.

Syntax

```
extern bb_Error_e
Ixf1110SetChipOnline(bb_ChipData_t *pChipData,
              bb_ChipSegment_t *section);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
ptSegment	bb_ChipSegment_t*	Chip section to set online, inclusive of channel

Returns

bb_Error_e	Error
b_NO_ERROR	Success

ixf1110SetChipOffline

Set one or more of the ports offline.

Syntax

```
extern bb_Error_e
Ixf1110SetChipOffline(bb_ChipData_t *pChipData,
              bb_ChipSegment_t *section);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
ptSegment	bb_ChipSegment_t*	Chip section or block to set offline inclusive of channel

Returns

bb_Error_e	Error
b_NO_ERROR	Success

ixf1110Write

Syntax

```
extern bb_Error_e
Ixf1110Write(bb_ChipData_t *pChipData,
              bb_Word_Size_t wordSize,
              ulong address,
              ushort length,
              void *buffer);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
wordSize	bb_Word_Size_t*	Enum size of data to be read: ONE_BYTE = 1 TWO_BYTES = 2 FOUR_BYTES = 4 EIGHT_BYTES = 8
address	ulong	Offset from chip base address to begin write
length	ushort	Number of words to write
buffer	Void*	Pointer to a structure that contains the data to be written

Returns

bb_Error_e	Error
b_NO_ERROR	Success

ixf1110GetBuildVersion

Returns information specific to the driver build.

Syntax

```
extern bb_Error_e
Ixf1110GetBuildVersion(bb_ChipData_t *pChipData,
                        char *drvName,
                        char *date,
                        ushort *buildVer,
                        ushort *buildRev);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
drvName	Char*	Character pointer to a buffer for the driver name
date	Char*	Character pointer to a buffer for the driver date
buildVer	ushort	Variable for build version
buildRev	ushort	Variable for build revision

Returns

bb_Error_e	Error
b_NO_ERROR	Success
drvName Char*	Character pointer to a buffer containing the driver name
date Char*	Character pointer to a buffer containing the driver date
buildVer ushort	Variable containing build version
buildRev ushort	Variable containing build revision

ixf1110InitAllocMemory

Allocates memory to support the driver data structures.

Syntax

```
extern bb_Error_e
Ixf1110InitAllocMemory(bb_ChipData_t *pChipData);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data

Returns

bb_Error_e	Error
b_NO_ERROR	Success

ixf1110DeAllocMemory

Deallocates the memory used to support the driver data structures.

Syntax

```
extern bb_Error_e
Ixf1110DeAllocMemory(bb_ChipData_t *pChipData);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data

Returns

bb_Error_e	Error
b_NO_ERROR	Success

ixf1110XgmacGetAddress

Syntax

```
extern bb_Error_e
Ixf1110XgmacGetAddress(bb_ChipData_t *pChipData,
                         bb_ChipSegment_t *section,
                         IxfApi_MacAddress_t *pMacAddress);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
section	bb_ChipSegment_t*	Chip block or section
address	Void*	Pointer to a buffer for the 48-bit MAC address to be read

Returns

Type	Description
bb_Error_e	Error
b_NO_ERROR	Success
address void*	Pointer to a buffer that contains the 48-bit MAC address read

ixf1110XgmacSetAddress

Function Definition

```
extern bb_Error_e
Ixf1110XgmacSetAddress(bb_ChipData_t *pChipData,
                         bb_ChipSegment_t *section,
```

```
IxfApi_MacAddress_t *pMacAddress);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
ptSection	bb_ChipSegment_t*	Chip block or section
address	Void*	Pointer to a buffer for the 48-bit MAC address to be written

Returns

bb_Error_e	Error
b_NO_ERROR	Success

3.1.3 The IXF1110 Device Specific Driver

3.1.3.1 Common Data Structure

The driver's common, or main, data structure is of the type bb_ChipData_t. This structure is used by most of the API routines at the IXF and devie layers. This main data structure encompasses the entire chip (in this case, the IXF1110) and alarm configuration. The common structure data members are listed in the following structure.

```
typedef struct  /* Complete Data for a Chip */
{
   bb_RegPointer_type  BaseAddress; /* Base Address of chip */
   bb_ChipType_e       ChipType;      /* Type of Chip */
   void*               pChipCfg; /* Pointer to Chip Specific Configuration */
   void*               pAlarmCfg; /* Pointer to Chip Specific Alarm Config */
   void*               funcPtr;       /* Pointer to the chip's api func's */
} bb_ChipData_t;
```

The structure ixf1110_ChipCfg_t is in the ixf1110_cnfg_d.h file. This structure is a member of bb_ChipData_t and is comprised of structures which contain configuration data for the different functions within the chip. For a complete list of ixf1110 data structures, refer ixf1110_xxx_d.h files (where xxx is the functional block within the chip).

```
typedef struct
{
     ixf1110_Spi4Cfg_t        Spi4Cfg;      /* SPI-4 block */
     ixf1110_RxCfg_t          RxCfg;        /* Global Rx block */
     ixf1110_TxCfg_t          TxCfg;        /* Global Tx block */
     ixf1110_SerDesCfg_t      SerDesCfg;    /* SerDes block */
     ixf1110_GlbStatusCfg_t   GlbStatusCfg; /*Global Status & config block*/
     ixf1110_GbicCfg_t        GbicCfg;      /*GBIC block*/
     ixf1110_XgmacCfg_t       XgmacCfg;     /*MAC control block*/
} ixf1110_ChipCfg_t;
```

3.1.3.2 Error Codes

The following tables contain a complete list of error codes returned by the driver.

Error Enumerator

```
typedef enum
{
  bb_NO_ERROR = 0,              /* Returned by Driver for no error */
```

Table 3-2. Fatal Error Types and Descriptions

Error	Description
bb_FATAL_ERROR	Fatal error codes should be defined here
bb_GENERAL_ERROR	Catch all type of error
bb_NULL_ADDRESS_ASSIGNED	A NULL Base Address has been assigned

Table 3-3. Common Error Types and Descriptions,,

Error	Description
bb_NO_CHIP_DATA = bb_COMMON_ERROR_OFFSET	pChipData = 0
bb_NULL_BASE_ADDR	BaseAddress = 0, for chip
bb_INV_BASE_ADDR	BaseAddress not for initialized chip
bb_INV_CHIP_TYPE	Chip type not supported
bb_NO_CHIP_CFG	pChipCfg = 0, for chip
bb_NO_ALARM_CFG	pAlarmCfg = 0, for chip, but Alarm cfg needed
bb_UNDEF_ALARM_BITS	An XxxAlarmCfg uses undefined alarm bits
bb_STM_MODE_MISMATCH	Mismatch between h/w and s/w cfg of STM-0/1
bb_INV_SEL_OH_BYTE	Invalid SelOhByte
bb_INV_SEL_OH_BYTES	Invalid SelOhBytes
bb_INV_SEL_COUNTERS	Invalid SelCounters
bb_INV_CHAN_TEST	Invalid Test Channel
bb_INV_PARAMETER	Invalid parameter, generic error
bb_INV_CHIP_SEGMENT	Chip segment is invalid
bb_NULL_ARG	null pointer passed as argument to function
bb_INV_BLOCK_OPERATION	operation not supported on this block
bb_FN_NOT_SUPPORTED	function not supported

Table 3-4. OHT Error Types and Descriptions ,

Error	Description
bb_JN_TRACE_WRITE_FAIL = bb_OHT_ERROR_OFFSET	Write of Expected/Rx/Tx J0,J1,J2 Trace failed
bb_INV_EXP_JN_FMT	The Expected Jn Format is invalid
bb_TX_J1_FOR_RPTR	Cannot set Tx J1 trace for repeater
bb_POH_PASSTHRU	Illegal call; all POH bytes passed through
bb_OHT_NOT_IN_TEST	OHT must be in test mode, to introduce errors
bb_TX_J0_NOT_CPU	For Terminal or ADM, Tx J0 source must = CPU
bb_INV_EXP_J1_FMT	An Invalid Expected J1 Format found
bb_INV_TX_J1_FMT	An Invalid Tx J1 Format found
bb_NO_OHT_NU_CFG	pNuBytes = 0

Error	Description
bb_NOT_PROTECTING_MAIN	not Main Terminal or ADM, or no Protection h/w
bb_NOT_TERM_ADM	This function valid only for Terminal or ADM
bb_INV_BKUP_OHT	Invalid Protection Allocation
bb_INV_SEL_OHT_CFG	Invalid SelOhtCfg Value Used
bb_TRACE_ACCESS_FAIL	Trace read or write has failed
bb_INV_TRACE_FORMAT	Invalid trace format selected
bb_INV_TRACE_TYPE	Invalid trace type selected

Table 3-5. Mapper Error Types and Descriptions,

Error	Description
bb_INV_PORT_NUM = bb_MAPPER_ERROR_OFFSET	Invalid PortNum
bb_NO_MPR_CFG	pMprCfg = 0
bb_INV_ALM_SPEC	Invalid AlmSpec
bb_NO_ADM_CFG	pAdmCfg = 0
bb_NOT_ADM	This chip not configured as an ADM
bb_INV_TIMESLOT	ADM timeslots = 0, or 1-28
bb_GET_J2_TYP_ERR	
bb_SET_J2_TYP_ERR	
bb_SET_J2_LEN_ERR,	
bb_PORT_NOT_N2_ENBLD	
bb_INV_SIG_LBL	
bb_INV_SEL_AUTO_FDBK	Invalid Auto Feedback spec
bb_INV_TEST_CNFG	
bb_Error_e	

3.1.3.3 Ixf1110 Device Driver

The IXF API device driver for the IXF1110 device is implemented in standard ANSI C, which allows the code to be highly portable. The code is fully re-entrant. The Linux kernel version under which this driver will function is 2.4. The user should familiarize him or her self with the IXF API User's Guide. This guide should be used as a companion to this document and it goes into more depth with the actual routines. The driver runs as a kernel module under Linux. The kernel module under Linux does not implement any communication to user mode space, it is to be used only by other kernel mode code. The user can either interface to the IXF API or the Device Specific Driver Layer in order to accomplish his goal.

The driver exports an interrupt handling routine (IxfApiChipIsr) to handle alarms. See the IxfApiChipIsr API section to learn how to handle the low level interrupts from the processor and call the IxfApiChipIsr, to handle the alarms within the device.

Most of the API routines accept a handle describing the type of device, where the device is located, and some configuration structures. Many API routines expect a "section" parameter, this tells the routine the location of the item the user wants to set or retrieve. An example of what should be populated in the section (of type bb_ChipSegment_t) is the channel number. Another expected parameter in many of the API routines will be an enumerated type applicable to the functionality of the routine. Again, the IXF API User's Guide should be referenced for more information. The following API descriptions attempts to highlight the possible values for each routine as well as alternative device specific routines the user can use.

IXFApiInit

This routine initializes the driver with the base address and chip type and returns a handle. API calls made after this call will use this handle. The valid value for the base address is the proper physical or virtual address of the device in the system and the chip type should be set to *bb_1110_CHIP*. Under Linux, the calling code is expected to pass the handle returned from *ioremap( )* for the correct virtual address.

IxfApiInitChip

This routine sends an internal RAM-based initialization table of register values to the device. The table is identified in the structure *bb_Chip_Data_t*, a pointer to which is passed to this function as an input parameter.

IxfApiAllocDataStructureMem

This routine dynamically allocates memory within the *pChip_Data* parameter for members that need dynamic allocation. The user calls this routine to enable alarm capabilities and chip initialization. This routine must be called before *IxfApiGetCfg*, *IxfApiSetCfg*, and *IxfApiSetAlarmCfg* are used. The *pChipData* pointer must be initialized via *IxfApiInit* prior to calling this function.

IxfApiDeAllocMemory

This routine frees all memory allocated during the *IxfApiAllocDataStructureMem* call.

IxfApiSetAlarmCfg

This routine is used to modify interrupt enable masks for a specific set of interrupts.

IxfApiReset

This routine resets all or some portion of the device. The alternative device specific driver would be *Ixf1110Reset*. The valid values for the *ResetType* enumeration parameter are:

```
bb_RESET_RX_FIFO
bb_RESET_TX_FIFO
bb_RESET_XGMAC
bb_RESET_CORE_CLK
bb_RESET_XGMAC_ALL
bb_RESET_RX_FIFO_ALL
bb_RESET_TX_FIFO_ALL
```

IxfApiGetChipInfo

This routine returns the chip related information. This routine retrieves a pointer to the *bb_ChipInfo_t* containing the chip id and chip version numbers. The alternative to this routine for the device specific driver would be *Ixf1110GetChipInfo*.

```
typedef struct              /* Information set by the hardware */
{
   uchar   ChipVersion;     /* Chip Version */
   ushort  ChipId;         /* Chip ID */
   ushort  VendorID        /*Vendor ID*/

} bb_ChipInfo_t;
```

IxfApiSetCfg

This routine is used to configure the chip. Many types of configurations can be performed with the combinations of *SetCfg*. The configuration is passed within the *pChipData* variable. Valid enum values for sections within the ifx1110 chip are:

- ixf_eXGMAC0
- ixf_eXGMAC1
- ixf_eXGMAC2
- ixf_eXGMAC3
- ixf_eXGMAC4
- ixf_eXGMAC5
- ixf_eXGMAC6
- ixf_eXGMAC7
- ixf_eXGMAC8
- ixf_eXGMAC9
- ixf_eGLBL
- ixf_eRX
- ixf_eTX
- ixf_eSPI4
- ixf_eSERDES
- ixf_eGBIC

IxfApiGetCfg

This routine is used to get the configuration of the chip. Many types of configurations can be performed with the combination of *GetCfg*. The chip configuration is returned within the *pChipData* variable. The same valid enum values for sections used in *IxfApiSetCfg* apply to this function as well.

IxfApiGetStatus

This routine returns status information. The status returned is comprised of register or register bit values that provide various status information from the device.

IxfApiGetCounters

This routine retrieves counter values. This can be in the form of a single counter or a group of counters. The alternative device specific routine is *Ixf1110GetCounters*. The enums for valid sections or blocks in the device where counter(s) may be retrieved are:

- ixf_eXGMAC0
- ixf_eXGMAC1
- ixf_eXGMAC2
- ixf_eXGMAC3
- ixf_eXGMAC4
- ixf_eXGMAC5
- ixf_eXGMAC6
- ixf_eXGMAC7
- ixf_eXGMAC8
- ixf_eXGMAC9
- ixf_eRX
- ixf_eTX

The valid individual counter request values are:

- ixf_eXGMACX:
  - bb_XGMAC_TX_OCT_OK_FRM_CNT
  - bb_XGMAC_TX_OCT_BAD_FRM_CNT
  - bb_XGMAC_TX_UC_FRM_CNT
  - bb_XGMAC_TX_MC_FRM_CNT
  - bb_XGMAC_TX_BC_FRM_CNT
  - bb_XGMAC_TX_64_OCT_FRM_CNT
  - bb_XGMAC_TX_65_TO_127_OCT_FRM_CNT
  - bb_XGMAC_TX_128_TO_255_FRM_CNT
  - bb_XGMAC_TX_256_TO_511_FRM_CNT
  - bb_XGMAC_TX_OK_512_TO_1023_CNT
  - bb_XGMAC_TX_OK_1024_TO_15XX_CNT
  - bb_XGMAC_TX_OK_15XX_TO_MAX_CNT
  - bb_XGMAC_TX_DEFERRED_CNT
  - bb_XGMAC_TX_TOTAL_COL_CNT

- bb_XGMAC_TX_SINGLE_COL_CNT
- bb_XGMAC_TX_MULTI_COL_CNT
- bb_XGMAC_TX_LATE_COL_CNT
- bb_XGMAC_TX_EXCESS_COL_ERR_CNT
- bb_XGMAC_TX_EXCESS_DEFER_RCV_CNT
- bb_XGMAC_TX_EXCESS_LEN_DROP_CNT
- bb_XGMAC_TX_UNDERRUN_CNT
- bb_XGMAC_TX_TAGGED_CNT
- bb_XGMAC_TX_CRC_ERR_CNT
- bb_XGMAC_TX_PAUSE_FRM_CNT
- bb_XGMAC_TX_FLOW_CTL_COL_SEND_CNT

- bb_XGMAC_RX_OCT_OK_FRM_CNT
- bb_XGMAC_RX_OCT_BAD_FRM_CNT
- bb_XGMAC_RX_UC_FRM_CNT
- bb_XGMAC_RX_MC_FRM_CNT
- bb_XGMAC_RX_BC_FRM_CNT
- bb_XGMAC_RX_64_OCT_FRM_CNT
- bb_XGMAC_RX_65_TO_127_OCT_FRM_CNT
- bb_XGMAC_RX_128_TO_255_FRM_CNT
- bb_XGMAC_RX_256_TO_511_FRM_CNT
- bb_XGMAC_RX_OK_512_TO_1023_CNT
- bb_XGMAC_RX_OK_1024_TO_15XX_CNT
- bb_XGMAC_RX_OK_15XX_TO_MAX_CNT
- bb_XGMAC_RX_FCS_ERR_CNT
- bb_XGMAC_RX_TAGGED_CNT
- bb_XGMAC_RX_DATAQ_ERROR_CNT
- bb_XGMAC_RX_ALIGN_ERR_CNT
- bb_XGMAC_RX_LONG_ERR_CNT
- bb_XGMAC_RX_JABBER_ERR_CNT
- bb_XGMAC_RX_PAUSE_RCV_CNT
- bb_XGMAC_RX_UNKNOWN_FRM_CNT
- bb_XGMAC_RX_VERY_LONG_ERR_CNT
- bb_XGMAC_RX_RUNT_ERR_CNT
- bb_XGMAC_RX_SHORT_ERR_CNT

  - bb_XGMAC_RX_CARRIER_EXT_ERR_CNT
  - bb_XGMAC_RX_SEQUENCE_ERR_CNT
  - bb_XGMAC_RX_SYMBOL_ERR_CNT
  - bb_XGMAC_COUNTERS /*All XGMACX counters. */
- ixf_eRX:
  - bb_RX_FRMS_REMOVED_0_CNT
  - bb_RX_FRMS_REMOVED_1_CNT
  - bb_RX_FRMS_REMOVED_2_CNT
  - bb_RX_FRMS_REMOVED_3_CNT
  - bb_RX_FRMS_REMOVED_4_CNT
  - bb_RX_FRMS_REMOVED_5_CNT
  - bb_RX_FRMS_REMOVED_6_CNT
  - bb_RX_FRMS_REMOVED_7_CNT
  - bb_RX_FRMS_REMOVED_8_CNT
  - bb_RX_FIFO_ERR_FRMS_DROP_CNT
  - bb_RX_FIFO_OVR_FRMS_CNT
- ixf_eTX:
  - bb_TX_FRMS_REMOVED_0_CNT
  - bb_TX_FRMS_REMOVED_1_CNT
  - bb_TX_FRMS_REMOVED_2_CNT
  - bb_TX_FRMS_REMOVED_3_CNT
  - bb_TX_FRMS_REMOVED_4_CNT
  - bb_TX_FRMS_REMOVED_5_CNT
  - bb_TX_FRMS_REMOVED_6_CNT
  - bb_TX_FRMS_REMOVED_7_CNT
  - bb_TX_FRMS_REMOVED_8_CNT
  - bb_TX_FRMS_REMOVED_9_CNT

IxfApiGenericRead

This routine will read data from the device from the offset specified. The routine is passed a pointer to a structure of type *bbChipData* to identify the chip base address and type. Additionally, *wordSize* identifies the number of bytes in a word, address is the offset from the base address, and *length* indicates the number of words to read. A void pointer to *buffer* into which the read data is placed is also passed to the routine.

IxfApiGenericWrite

This routine will write data to the device from the offset specified. The parameters are the same as the *IxfApiGenericRead* with the exception that *buffer* contains data to be written to the indicated address.

IxfApiInitAlarmCallback

This routine registers a callback with the driver. The callback will be called whenever any alarm occurs in the system. The argument for this routine is a function pointer pointing to the callback function.

IxfApiChipIsr

This routine handles alarms. Its only parameter is the handle.

IxfApiGetMacAddress

This routine retrieves the 48-bit MAC address.

IxfApiSetMacAddress

This routine sets the 48-bit MAC address.

3.1.4 Kernel Mode ISR Driver

A separate loadable kernel module is supplied as part of the IXD2810 Linux device driver for the purpose of hardware interrupt support. The interrupt driver is the *ixd2810IntMod.o* and is kept separate from the device access driver in order to maintain platform independence. All platform specific code for the IXD2810 media card is in the *ixd2810IntMod.o* module. The application first performs the call to the *IxfApiInit( )* function to obtain the *bb_ChipData_t* handle from the IXF driver for the device in question. This handle is then passed to the *ixd2810IntMod.o* module via the call to IXD2810 ISR initialization function:

```
int ixd2810IsrInit
    (
    bb_ChipData_t *pChipData/* */
    )
```

The call to ixd2810IsrInit will connect the driver to the interrupt via OS provided calls and LSP provided information about the hardware specifics.Once initialized, the *ixd2810IntMod.o* module will serve as the ISR for the device and service all hardware interrupts. Once an interrupt occurs and is serviced, the *ixd2810IntMod.o* module will perform a call to the *IxfApiChipIsr( )* function in the device access driver.

3.1.5 IXD2810 Driver Unit Tests

These tests will be conducted only on the IXDP2810 daughter card independent of rest of the system. These tests will be considered complete if all tests yield a "PASSED" result.

IxfApiInit Test

- Configure IXFAPI driver for the IXF1110 device.
- Tests return status of the routine.

IxfApiInitChip Test

- Initializes the device with a certain configuration.
- Tests return status of the routine.

IxfApiAllocDataStructureMem Test

- Configure IXF API driver to allocate any necessary memory needed by the device driver.
- Tests return status of the routine.

IxfApiDeAllocMemory Test

- Configure IXF API driver to deallocate any memory allocated during the IxfApiAllocDataStructureMem.
- Tests return status of the routine.

IxfApiGetStatus Test

- Retrieves a certain status values from the device functional blocks.
- Tests return status of the routine.

IxfApiReset Test

- Writes a certain configuration to the device using IxfApiInitChip then uses this Reset routine to reset the entire device. Once this is done, a comparison of the known default values is done.
- Tests return status of the routine.

IxfApiInitAlarmCallback Test

- Initializes the alarm callback mechanism.
- Tests return status of the routine.

IxfApiChipIsr Test

- Retrieves chip ISR status from the device.
- Tests return status of the routine.

IxfApiGetCounters Test

- Retrieves a certain counter value from the device.
- Tests return status of the routine.

IxfApiGetChipInfo Test

- Retrieves the IXD2810 board ID from the device and compares the value to what is expected.
- Tests return status of the routine.

IxfApiSetChipOnline Test

- Brings some or all of the ports online.
- Tests return status of the routine.

IxfApiSetChipOffline Test

- Brings some or all of the ports offline.
- Tests return status of the routine.

IxfApiSetAlarmCfg Test

- Edits the interrupt enable masks in the device.
- Tests return status of the routine.

IxfApiGetBuildVersion Test

- Retrieves the build version from the driver and compares the value to what is expected.
- Tests return status of the routine.

IxfApiGenericRead Test

- Retrieves the current value in the same location as what the IxfApiGenericWrite wrote to and compare the value
- Tests return status of the routine.

IxfApiGenericWrite Test

- Writes a certain value to a certain location then the GenericRead will compare.
- Tests return status of the routine.

IxfApiSetCfg Test

- Writes a certain configuration table to a certain block, then IxfApiGetCfg will be called to read the values back.
- Tests return status of the routine.

IxfApiGetCfg Test

- Used in IxfApiSetCfg Test above.
- Tests return status of the routine.

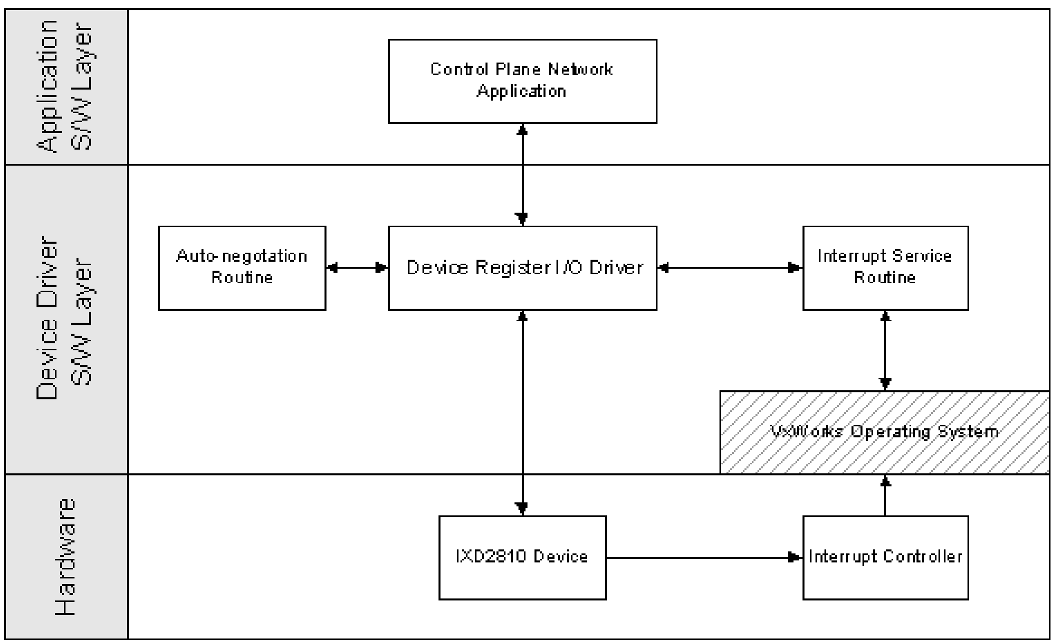
3.2 VxWorks Environment

The IXD2810 device driver discussed in this section is for the VxWorks* operating system on the Intel® IXDP2800 Advanced Development Platform. The device driver for the IXDP2810 is implemented as a downloadable module for the VxWorks environment.

3.2.1 Design Decomposition

Figure 3-3 shows the design of the device drivers and the environment in which the driver executes. The figure also shows the major components used in the design, and the relationship among those components:

Figure 3-3. Software Architecture Block Diagram



The blocks shown in the hardware level include system hardware controllers used by the IXD2810 media card as well as the media card itself. The modules, shown in the device driver software layer form the IXD2810 Device Driver Library, which provides the driver interface to the hardware.

The functions in the application layer are typical network applications that use the APIs provided by the device driver to communicate with the hardware functions. The IXD2810 configuration driver is implemented as a library. The application can access the IXD2810 card by communicating with the Device Register function.

The device register I/O driver function is a classic I/O driver for the IXD2810 register address space. This function provides the I/O control library for initialization and configuration of the MAC device. The library defines functions, such as `ixd2810_Start(), ixd2810_Stop()`, and `ixd2810_Ioctl()`, available to the network application program to configure and access IXD2810 media card.

The auto-negotiation I/O function configures the link ports on start-up. This function detects the capabilities of the other devices (over a common link), and it adjusts its transmit and receive (Tx and Rx) signals to match those of a link partner. The device driver handles the auto-negotiation privately, which appears transparent to the network application program.

The interrupt service routine handles the interrupts mapped to the NPU interrupt space. The interrupts are wire-or'ed together in the hardware, which directs the architecture of the interrupt service routine to identify the interrupt type from the IXD2810 registers. The interrupt service function provides the mechanism to respond to different types of interrupts and also provides queues to service the requests.

3.2.1.1 Hardware Layer

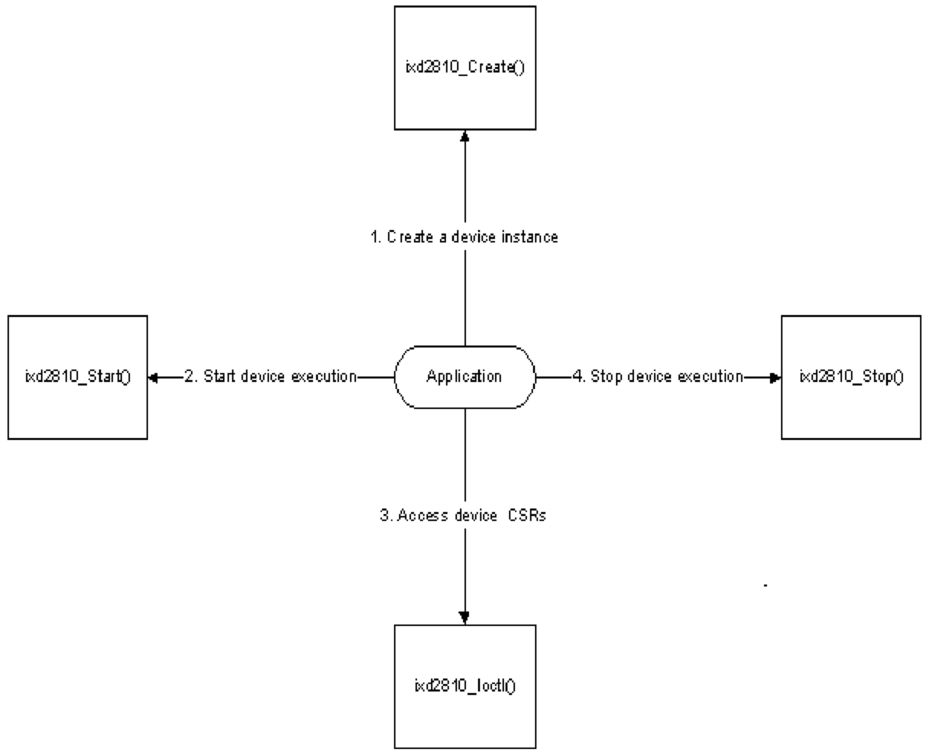
There are only two hardware modules shown in the context to the software architecture. Since these are the only modules that provide an interface from the NPU to the IXD2810 media card, the memory controller provides the physical address mapping of the IXD2810 register block to the NPU address space. The interrupt controller provides the mapping of the IXD2810 interrupts to the NPU interrupt map.

3.2.2 External APIs

The device driver provides the application interfaces to the IXD2810 Media Card for VxWorks platform that runs on the Intel® IXDP2800 Advanced Development Platform. The functions of the driver are used to access the IXD2810 registers. Figure 3-4 shows the calling sequence from the application to the device library functions. The label describes the functionality of each procedure. Each label contains a number that increases counterclockwise. This number represents the step in which each function should be called. For example, `ixd2810_Create()` should be the first function called and `ixd2810_Stop()` the last.

The driver application entry points for the IXD2810 Media Card and module implementations are described in the following sections.

Figure 3-4. Function Calling Sequence



3.2.3 Data Structures

The device driver maintains the state of every device using an array of the IXD2810_DEV data structure. The driver accesses a particular device structure in the device array by using the device ID. The content of the IXD2810_DEV structure and the device array are shown as follows.

```
typedef struct _ixd2810_dev
{
     DEV_HDRdevHdr;
     uint32_tdevId; /* device ID */
     uint32_t vbase; /* virtual address base */
     uint32_tgbicStatus;/* GBIC Status */
     BOOL created; /* TRUE if this device has been created */
     BOOL isRunning; /* device current status/mode */
     int intCnt; /* interrupt count */
     int intLevel; /* interrupt level */
     int taskId; /* task ID spawn */
     SEM_ID mutexSem; /* mutex semaphore */
     SEM_ID syncSem; /* sync semaphore */
     DEV_PORTS ports[N_PORTS];/* ports on device */
} IXD2810_DEV;

     LOCALIXD2810_DEV ixd2810Dev[N_DEVICES];
```

3.2.3.1 ixd2810_Create( )

This function performs the creation and initialization of the device structure specified by the device identification, devId. The function queries the system for the presence of the IXD2810 card via the board ID register. If the IXD2810 card is not found, then an error is returned. Otherwise, the function proceeds to create and initialize the device structure for corresponding to the device `devId`.

```
STATUS ixd2810_Create(uint32_t devId);
```

The function body pseudocode is as follows:

- Verify that the devId is valid device identification. Otherwise, return an error.
- Get a pointer to the device structure indexed by the `devId`.
  ```
  pDev = &ixd2810Dev[devId];
  ```
- Verify that the device has not previously been created. If so, return an error.
- Retrieve the board ID and verify that it is valid. If it is not valid, return an error.
- Initialize the device structure internal variables. This step includes, storing a copy of the device ID, storing the name assigned to the device, assign the interrupt level to this device, and created a semaphore for synchronization purposes and save it in the device structure.

3.2.3.2 ixd2810_Start( )

This function performs the execution process of the specified IXD2810 device, `devId`. This function completes auto-negotiation for all ports, and installs the IXD2810 device interrupt handler. This function should not be called again without first calling the `ixd2810_Stop()` function.

```
STATUS ixd2810_Start(uint32_t devId);
```

The function body pseudocode is as follows:

- Verify that the `devId` is a valid device identification. Otherwise, return an error.
- Get a pointer to the device structure indexed by the `devId`.
  ```
  pDev = &ixd2810Dev[devId];
  ```
- Verify that the device has been created. If it has not been created, return an error.
- Verify that the device is not executing. If it is executing, return an error.
- Perform auto-negotiation for each port.
  ```
  for(p = 0; p < N_PORTS; p++)
  if (doAutoneg(pDev, p) == ERROR)
  return ERROR;
  ```
- Install the interrupt handler.
  ```
  if (ixdp2810IntDevInit(pDev) == ERROR)
  return ERROR;
  ```

3.2.3.3 ixd2810_Stop( )

This function disables the port receive and transmit operations, and resets the SPI4-3 and GBIC interfaces for the device devId. This function should be called after the `ixd2810_Start()` has been successfully called. Otherwise, `ixd2810_Stop()` will return without making any changes to the MAC device.

```
STATUS ixd2810_Stop(uint32_t devId);
```

The function body pseudocode is as follows:

- Verify that the `devId` is valid device identification. Otherwise, return an error.
- Get a pointer to the device structure indexed by the devId.
  ```
  pDev = &ixd2810Dev[devId];
  ```
- Verify that the device is executing. If false, return an error.
- Release the interrupt handler.
  ```
  ixd2810IntDevRelease(pDev);
  ```
- Delete the device semaphore.
  ```
  semFlush(pDev->syncSem);
  semDelete(pDev->syncSem);
  ```

3.2.3.4 ixd2810_Ioctl( )

The device driver provides an I/O control function to the application program for accessibility to the IXD2810 registers. presents the `ixd2810_Ioctl()` function declaration.

This function is the entry point to perform configuration and get status of the device. This function can only be called once the IXD2810 has been started, that is, after the `ixd2810_Start()` function has been successfully called. This function performs the different functions accesses based upon the `cmd` parameter.

The first parameter, `devid`, expected by the `ixd2810Ioct()`, identifies the IXD2810 card. The second parameter determines the type of command being requested by the application program. The device library defines two basic I/O commands, namely, read and write. The application program can request a read, or get, instruction from the device by passing the `IXD2810_IOCTL_GET` macro to `ixd2810_Ioctl()`. Similarly, to request a write, or put, instruction to the device, the application should pass the `IXD2810_IOCTL_SET` macro to `ixd2810_Ioctl()`.

The third parameter, `stBlock`, provides the specific of the I/O control command.

```
STATUS IXD2810Ioctl(
uint32_t devId,
uint32_t cmd,
block_t *stBlock
```

The function body pseudocode is as follows:

- Verify that the devId is valid device identification. Otherwise, return an error.
- Get a pointer to the device structure indexed by the devId.
  ```
  pDev = &ixd2810Dev[devId];
  ```
- Verify that the device has been created and is executing. If false, return an error.
- Call the corresponding function depending on the cmd parameter. The following sections describe the corresponding function bodies for the `ixd2810IoctGet()` and `ixd2810IoctSet()`.

```
switch(cmd)
{
 case IXD2810_IOCTL_GET:
   status = ixd2810IoctlGet(pDev, stBlock);
   break;
 case IXD2810_IOCTL_SET:
   status = ixd2810IoctlSet(pDev, stBlock);
```

```
      break;
   default:
      (void) errnoSet (S_ioLib_UNKNOWN_REQUEST);
      status = ERROR;
      break;
    }
```

Example of ixd2810IoctlGet( )

```
switch(stBlock->blockId)
   {
     case PORT0_BLOCK thru PORT9_BLOCK:
     case GLOBAL_BLOCK:
     case TX_BLOCK:
     case RX_BLOCK:
     case SPI42_BLOCK:
     case SERDES_BLOCK:
       for(regOffset = 0; regOffset < stBlock->regNum; regOffset++)
       IXD2810_REG_GET(pDev->vbase +
          BLOCK_REG_ADRS(stBlock->blockId, stBlock->startingReg +
         regOffset),
          stBlock->data[regOffset]);
       break;
     case GBIC_BLOCK:
       for(regOffset = 0; regOffset < stBlock->regNum; regOffset++)
          IXD2810_REG_GET(pDev->vbase + GBIC_BASE +

        stBlock->startingReg + regOffset, stBlock->data[regOffset]);
     break;
     case BOARDID_BLOCK:
       IXD2810_REG_GET(pDev->vbase + IXD2810_BOARDID_OFFSET,

     stBlock->data[1]);
                  break;
         default:
         Block is not defined.
```

Example of ixd2810IoctlSet( )

```
/* access registers indexed by blocks on the memory map */
switch(stBlock->blockId)
{
   case PORT0_BLOCK thru PORT9_BLOCK:
   case GLOBAL_BLOCK:
   case SPI42_BLOCK:
   case SERDES_BLOCK:
    for(regOffset = 0; regOffset < stBlock->regNum; regOffset++)
     {
      IXD2810_REG_SET(pDev->vbase +
       BLOCK_REG_ADRS(stBlock->blockId, stBlock->startingReg +
        regOffset),
                   stBlock->data[regOffset]);
     }
     break;
   case GBIC_BLOCK:
    for(regOffset = 0; regOffset < stBlock->regNum; regOffset++)
     {
      IXD2810_REG_SET(pDev->vbase + GBIC_BASE +
                         stBlock->startingReg + regOffset,
                     stBlock->data[regOffset]);
     }
```

```
    break;
   case TX_BLOCK:
   case RX_BLOCK:
   case BOARDID_BLOCK:

 Block is read-only; unable to modify its content.

   default:
      Block is not defined.
}
```

The `BLOCK_REG_ADRS` macro calculates the absolute memory block location within the memory map using the given block b and register r. The macro is defined as follows:

```
#define BLOCK_REG_ADRS(b, r) (((b & 0xf) << 7) + r)
```

3.2.4 System Components

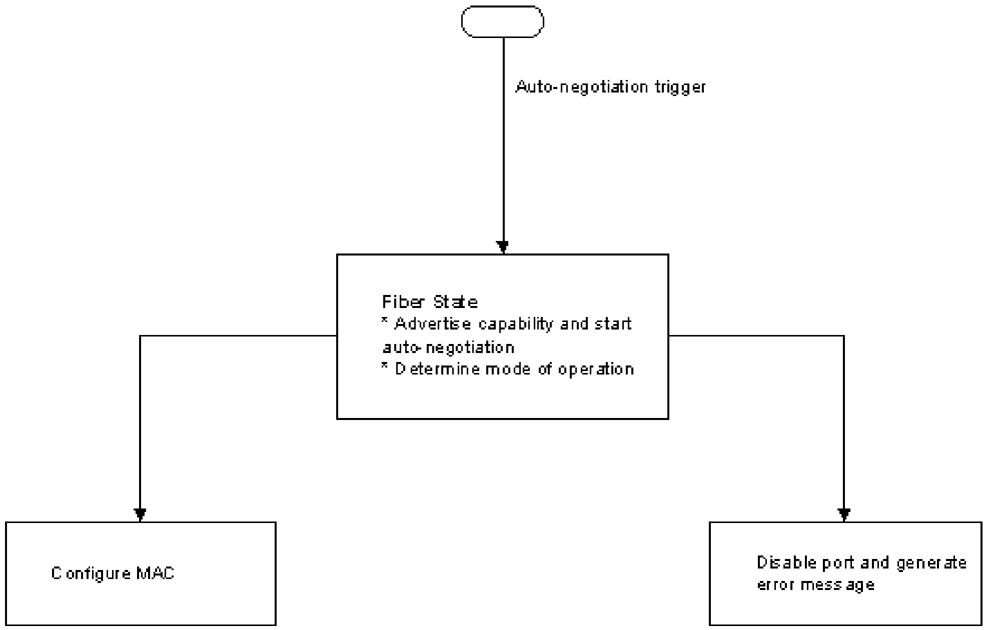
3.2.4.1 Auto-Negotiation

The auto-negotiation function drives the auto-negotiation on the IXF1110 Ethernet MAC in fiber mode. Figure 3-5 demonstrates the sequence of operation functions in auto-negotiation.

The IXD2810 only supports fiber mode Ethernet, so when auto-negotiation is triggered, the system is started in fiber mode and the fiber mode algorithm is initiated for the MAC device ports on the board. In fiber mode:

- Advertise capability and start auto-negotiation.
- Determine the mode of operation.
- If the mode is supported, configure IXF1110. If the mode is not supported, disable port and generate error message.

Figure 3-5. Auto-Negotiation Event Diagram



3.2.4.2 doAutoneg

(IXD2810_DEV *pDev, uint32_t port)

```
   fiberAutoNeg(pDev, port);/* fiber auto-negotiation */
   /* Fiber mode only supports giga speed and full duplex */
   if (pDev->ports[port].link == UP) {/* link up */
      if ((pDev->ports[port].speed == SPEED_1000) &&
      (pDev->ports[port].duplex == FULL))
         configMac(pDev, port); /* Configure MAC */
      else
         disablePort(pDev,port); /* Disable port */
   }
   else /* link down */
      disablePort(pDev, port);
```

void fiberAutoNeg

(IXD2810_DEV *pDev, uint32_t port)

```
   /* Start auto-negotiation */
   IXD2810_REG_GET(vbase + BLOCK_REG_ADRS(port, DIV_CONFIG_WORD),
  divData);
   divData |= DC_Autoneg_Enable;
   IXD2810_REG_SET(vbase + BLOCK_REG_ADRS(port, DIV_CONFIG_WORD),
  divData);

   /* Poll Rx Config Word to determine when AN is complete or timeout */
   for(delay = 0; delay < AN_TIMEOUT; delay++) {
     delayUSec(10000); /* 10 msec delay */
     IXD2810_REG_GET(vbase + BLOCK_REG_ADRS(port, RX_CONFIG_WORD),
```

```
 rxData);
   if (rxData & RC_An_Complete)
      break;
}

if (delay < AN_TIMEOUT) {
   /* Speed is fixed at giga */
   pDev->ports[port].speed = SPEED_1000;

   /* Get port status from Rx Config Word */
   if (rxData & RC_Half_Duplex)
       pDev->ports[port].duplex = HALF;
   if (rxData & RC_Full_Duplex)
       pDev->ports[port].duplex = FULL;
 /* Link is up when autoneg is enabled and finished */
   pDev->ports[port].link = UP;
 }
else { /* timeout has occurred */
   pDev->ports[port].link = DOWN;
   }
```

void configMac

(IXD2810_DEV *pDev, uint32_t port)

```
/* Set link up*/
IXD2810_REG_GET(pDev->vbase + LINK_LED_ENABLE, data);
data |= LINK_UP(port);
IXD2810_REG_SET(pDev->vbase + LINK_LED_ENABLE, data);

/* Enable port */
IXD2810_REG_GET(pDev->vbase + PORT_ENABLE, data);
data |= PORT_EN(port);
IXD2810_REG_SET(pDev->vbase + PORT_ENABLE, data);
```

void disablePort

(IXD2810_DEV *pDev, uint32_t port)

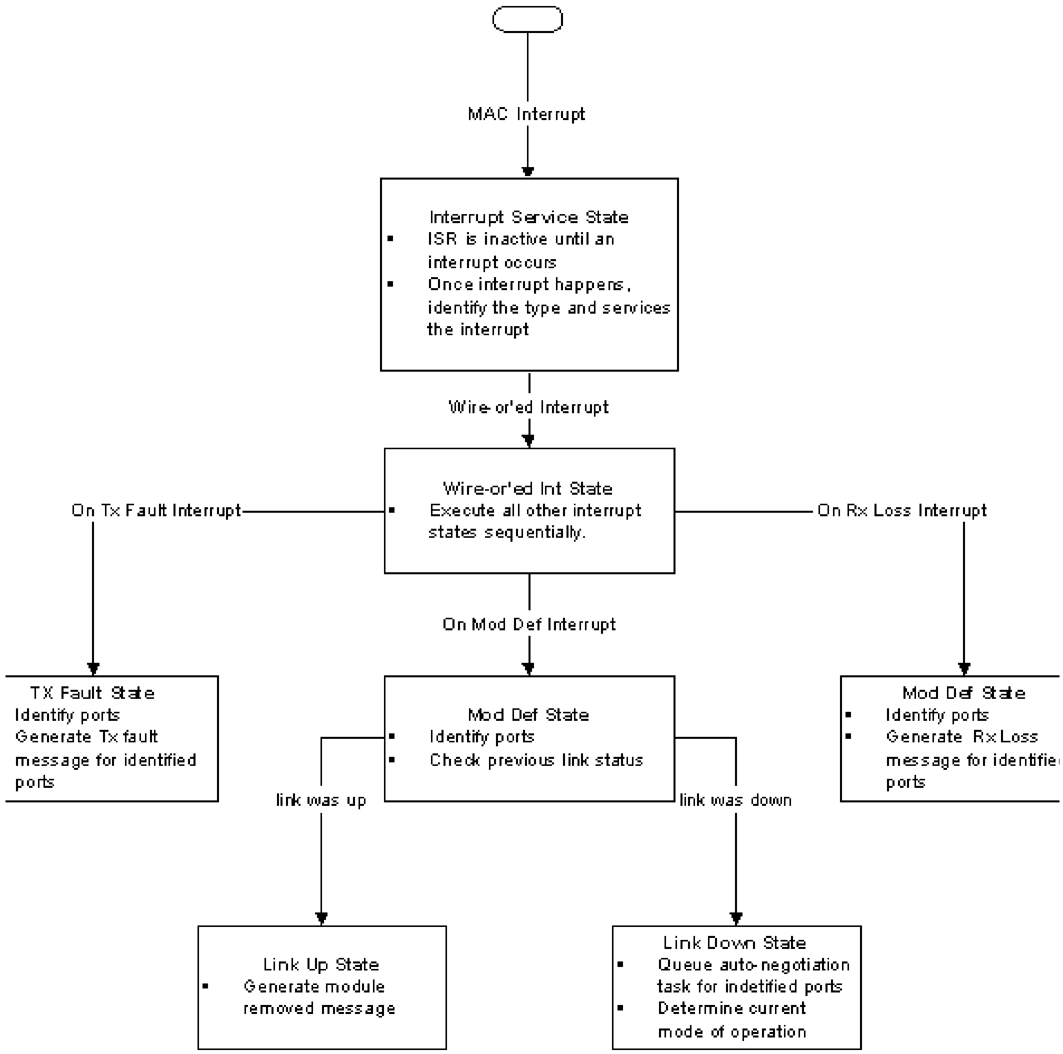
```
IXD2810_REG_GET(pDev->vbase + PORT_ENABLE, data);
data &= (ENABLE_BOUNDARY - PORT_EN(port));
IXD2810_REG_SET(pDev->vbase + PORT_ENABLE, data);

IXD2810_REG_GET(pDev->vbase + LINK_LED_ENABLE, data);
data &= (ENABLE_BOUNDARY - LINK_UP(port));
IXD2810_REG_GET(pDev->vbase + LINK_LED_ENABLE, data);
```

3.2.4.3 Interrupt Service Routine

The interrupt service routines are provided to handle specialized interrupts that are mapped to the NPU address in the reference platform. The interrupts supported are TX_FAULT, RX_LOSS, and MOD_DEF as shown in Figure 3-6. The structure of the interrupt operation is shown in the state diagram as follows:

Figure 3-6. Interrupt Service Routine Diagram



The service routine is in an inactive state until a MAC interrupt gets trigged. Once an interrupt is generated, the interrupt is identified and appropriate state transition takes place according to the interrupt. In each state, the interrupt is serviced as specified in the state. The RX_LOSS_INT and TX_FAULT_INT are forwarded to the service state. The MOD_DEF_INT is serviced such that for any identified port, the state of the link for the selected port is checked and accordingly switched to either link up or down state. The wire-or'ed interrupt is serviced by executing all three interrupt states in sequence. Once any state terminates, the system return to interrupt wait state. Work done in each state is mapped to an independent function, as shown in the following table:

Table 7. Interrupt Service Routines

Interrupt Name	Handler Function	Description
RX_LOSS_INT	`void ixd2810RxLossIsr ( IXD2810_DEV *pDev, uint32_t irq, uint32_t status )`	Generates RX loss message for the identified device and port.
TX_FAULT_INT	`void ixd2810TxFaultIsr ( IXD2810_DEV *pDev, uint32_t irq, uint32_t status )`	Generates TX loss message for identified device and port.
MOD_DEF_INT	`void ixd2810ModDefIsr ( IXD2810_DEV *pDev, uint32_t irq, uint32_t status )`	The sequence of operation of the service routine is as follows: - If module was present before interrupt, display “module removed on port.” - If module was not present before interrupt, start auto-negotiation on port.

The body descriptions for all functions forming the interrupt handler module in VxWorks environments is as follows:

STATUS ixd2810IntDevInit(IXD2810_DEV *pDev)

```
   /* launch a task to serve the mod def irq at task level */
   taskId = taskSpawn(pDev->devHdr.name, TASK_PRIORITY,
                TASK_OPTS, TASK_STACK_SIZE, (FUNCPTR) ixd2810ModDefIsr,
                (int) pDev, 0, 0, 0, 0, 0, 0, 0, 0, 0);

   pDev->taskId = taskId;

   /* connect isr to interrupt handler */
   (void) intConnect((VOIDFUNCPTR*) INUM_TO_IVEC(pDev->intLevel),
      (VOIDFUNCPTR) ixd2810Intr, (int) pDev);

   /* enable interrupt */
   intEnable(pDev->intLevel);

   /* Enable interrupt in GBIC control register */
   IXD2810_REG_SET(pDev->vbase + GBIC_CONTROL, GC_INT_ENABLE);

   /* Read GBIC status value */
   IXD2810_REG_GET(pDev->vbase + GBIC_STATUS, pDev->gbicStatus);

STATUS ixd2810IntDevRelease(IXD2810_DEV *pDev)

   /* Disable interrupt in GBIC control register */
   IXD2810_REG_SET(pDev->vbase + GBIC_CONTROL, GC_INT_DISABLE);
```

```
    intDisable(pDev->intLevel);
    taskDelete(pDev->taskId);

void ixd2810Intr(IXD2810_DEV *pDev)

    /* Read GBIC status register */
    IXD2810_REG_GET(pDev->vbase + GBIC_STATUS, gstatus);

    /* Find which bit(s) changed */
    irq = gstatus ^ pDev->gbicStatus;

    if (irq & GS_Rx_Loss)
       {
      ixd2810RxLossIsr(pDev, irq, gstatus);
      irqServiced = TRUE;
       }
    if (irq & GS_Tx_Fault)
       {
      ixd2810TxFaultIsr(pDev, irq, gstatus);
      irqServiced = TRUE;
       }
    if (irq & GS_Mod_Def)
       {
      semGive(pDev->syncSem);
      irqServiced = TRUE;
       }

void ixd2810RxLossIsr(IXD2810_DEV *pDev, uint32_t irq, uint32_t status)

    for (port = 0; port < N_PORTS; port++, rxBitIndex <<= 1)
    {
       if (irq & rxBitIndex)
           { /* bit changed for this port */
           if (status & rxBitIndex)
               disablePort(pDev, port);
       else
          enablePort(pDev, port);
          }
    }

void ixd2810TxFaultIsr(IXD2810_DEV *pDev, uint32_t irq, uint32_t status)

    for (port = 0; port < N_PORTS; port++, txBitIndex <<= 1)
    {
       if (irq & txBitIndex)
          { /* bit changed for this port */
       if (status & txBitIndex)
          dprintf("Tx fault appeared on MAC%d port%d.\n",
            pDev->devId, port);
       else
          dprintf("Tx fault disappeared on MAC%d port%d.\n ",
```

```
            pDev->devId, port);
          }
        }

void ixd2810ModDefIsr(IXD2810_DEV *pDev)

   FOREVER
   {
   /* exclusive access */
   semTake(pDev->syncSem, WAIT_FOREVER);

   /* Read GBIC status register */
   IXD2810_REG_GET(pDev->vbase + GBIC_STATUS, status); /* bits 9:0 */

   /* Find which bit(s) changed */
   irq = status ^ pDev->gbicStatus;

   for (port = 0; port < N_PORTS; port++, modBitIndex <<= 1)
      {
      if (irq & modBitIndex) /* bit changed for this port */
         {
         /* if bit=1, module is not present */
         if (status & modBitIndex)
         {
         disablePort(pDev, port);
         autoFlag[pDev->devId][port] = 0;
         }
         else
          {    /* if bit=0, module is present */
            autoFlag[pDev->devId][port] = 1;
          }
        }
       }

   /* auto-negotiation each port */
   for (port = 0; port < N_PORTS; port++)
    {
    if (autoFlag[pDev->devId][port] == 1)
     {
     /* Start auto-negotiation */
        /* Put in immediate task queue */
        queue_task(&stModDefTask, &tq_immediate);
        /* Muse mask as bottom half */
        mark_bh(IMMEDIATE_BH);
     /* Clear auto-negotiation flag */
     autoFlag[pDev->devId][port] = 0;
      break;
     }
   }
  } /* FOREVER */
```

void ixd2810RxLossIsr

(IXD2810_DEV *pDev, uint32_t irq, uint32_t status)

void ixd2810TxFaultIsr

(IXD2810_DEV *pDev, uint32_t irq, uint32_t status)

void ixd2810ModDefIsr

(IXD2810_DEV *pDev)

```
    /* Read GBIC status register */
IXD2810_REG_GET(pDev->vbase + GBIC_STATUS, status); /* bits 9:0 */

/* Find which bit(s) changed */
irq = status ^ pDev->gbicStatus;

for (port = 0; port < N_PORTS; port++, modBitIndex <<= 1)
{
    if (irq & modBitIndex) /* bit changed for this port */
     {
      /* if bit=1, module is not present */
      if (status & modBitIndex)
       {
       disablePort(pDev, port);
       autoFlag[pDev->devId][port] = 0;
       }
      else
       { /* if bit=0, module is present */
         autoFlag[pDev->devId][port] = 1;
       }
     }
}

/* auto-negotiation each port */
for (port = 0; port < N_PORTS; port++)
    {
    if (autoFlag[pDev->devId][port] == 1)
       {
         /* Put in immediate task queue */
         queue_task(&stModDefTask, &tq_immediate);
         /* Muse mask as bottom half */
         mark_bh(IMMEDIATE_BH);
        /* Clear auto-negotiation flag */
        autoFlag[pDev->devId][port] = 0;
        break;
       }
 }
```

void getModState

(void *ptr)

```
/* Retrieve task data */
ClientData *stData = (ClientData *)ptr;
pDev = stData->pDev;
port = stData->port;

/* do auto-negotiation on current port */
doAutoneg(pDev, port);

/* Queue the next auto-negotiation task */
while (++port < N_PORTS) {
   if (iAutoFlag[port] == 1) {
         stDevData.iDev = iDev;
         stDevData.port = port;
         queue_task(&stModDefTask, &tq_immediate);
         mark_bh(IMMEDIATE_BH);
         iAutoFlag[port] = 0;
         break;
   }
}
```

Single OC-192 I/O Card 4

4.1 System Overview

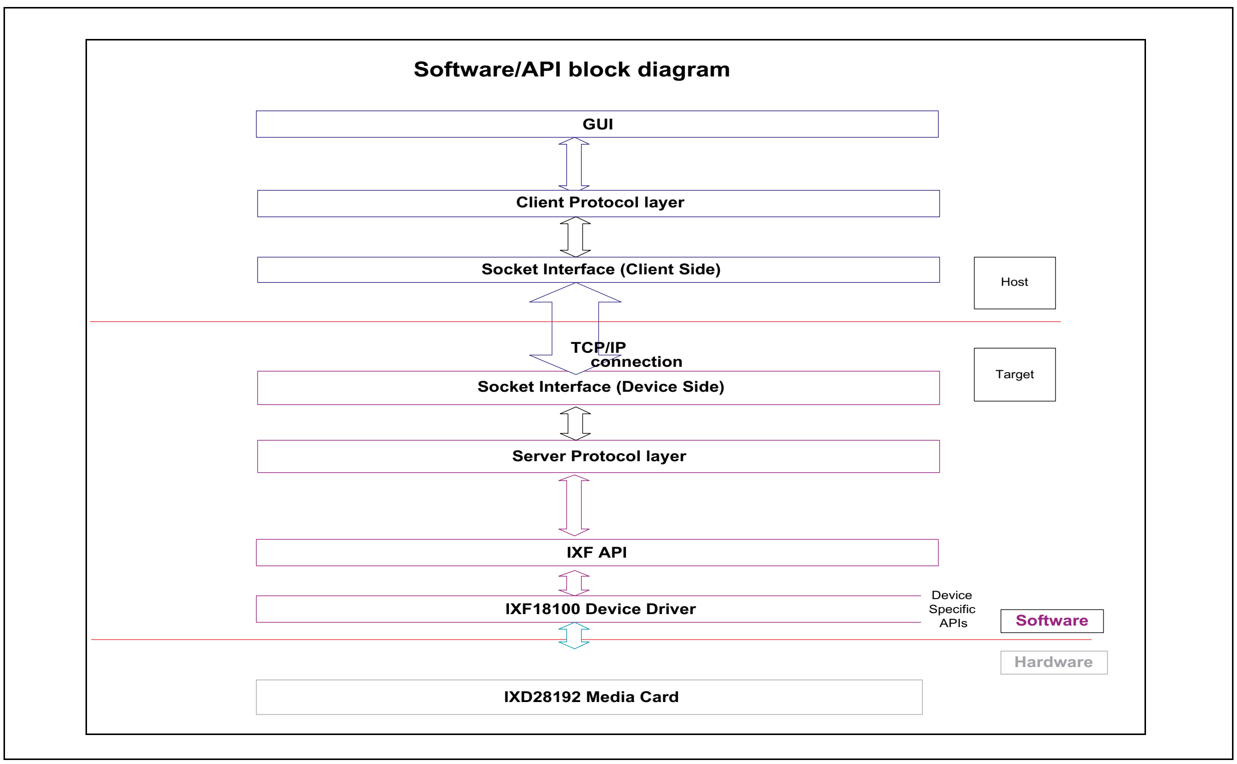
The IXD28192 is a single-channel, full-duplex, OC-192 Packet over SONET (POS), Ethernet LAN, and Ethernet WAN media board for the IXDP2800.

The IXDP28192 line card plugs into the main board of the IXDP2800. The IXDP28192 consists of a media interface to the network processors of IXDP2800, the IXF18100 framer, and the line interface to connect to transmit and receive optical fibers. The device driver is implemented as a downloadable module for both the VxWorks and Linux environments.

4.1.1 Design Decomposition

The IXDP28192 device driver and the supporting software is designed to be modular and portable. There is no need for synchronization between the Master and Slave NPUs.

Figure 4-1. Device Driver Design



The device driver for the IXDP28192 has the following components:

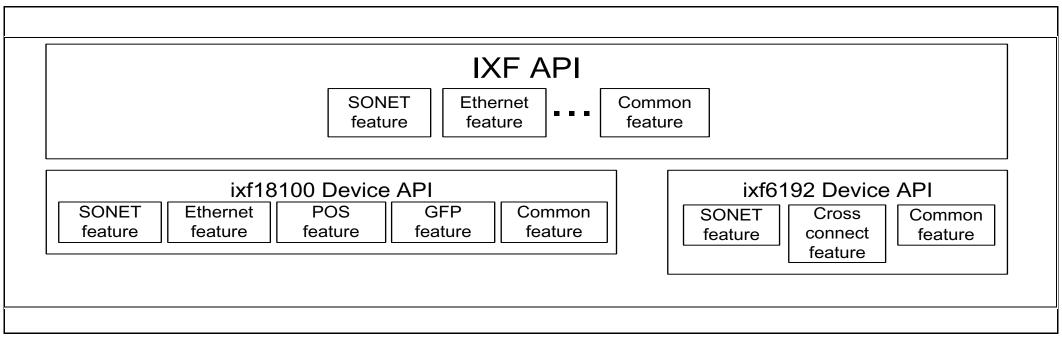
- IXF API Module
- The IXF18100 Device Specific Driver

- Intel Optical Component Management Software (OCMS) - Graphical User Interface (GUI) Configuration Tool
- Proprietary Protocol System
- Communication Server

4.2 IXF API Module

The IXF API provides a common and consistent interface for supported IXF devices. For each supported device, the IXF API provides device specific functions, as well as common interface functions for features that are not specific to a particular device. The following diagram illustrates the IXF API, feature API, and device API layers.

Figure 4-2. IXF API Model



The IXF API provides IXF18100 specific functions for general device configuration, while access to the POS, Ethernet, and other functional blocks of the IXF18100 is provided by interface functions.In addition, the IXF API also provides generic read and write access to the devices.

The IXF API interfaces to multiple devices (whether of the same or different types) simultaneously. The IXF API provides the application with a 'chip ID' that is used to reference the device it is accessing.

4.3 Feature APIs

Each 'feature' has its own API, and provides a common interface to the functionality of different devices. Each device may have its own implementation of a feature API, as the implementation may differ from device to device.

Each Feature API function signature must match the corresponding IXF API function exactly, as it is a Feature API function that actually provides the implementation for the IXF API function. As a result, the application has a common interface to functionality that is shared by several devices. Feature APIs are independent of each other; as for some devices (IXF18100, for example) subsets of the complete feature set must work properly.

There are different types of Feature APIs:·

Table 4-1. Feature API Types

Type	Description
Device specific	Specific to a device. Examples of this include global registers and global configuration.
Common	Common to all (or at least the vast majority of) devices. Functionality includes resetting the device, getting the device ID/version, and generic read and write access.
Functionality based	Provides interfaces for the following blocks: POS GFP SONET SPI4 ATM Ethernet PCS

4.3.1 Device APIs

Each device has its own API that provides device-specific implementation of IXF API functions. The Device APIs could be described as 'composite' APIs, as each Device API consists of device-specific functions, plus the feature APIs for all features that the device supports. The Device API function signatures must match the corresponding IXF API functions exactly, as it is a device API function that actually provides the implementation for almost all IXF API functions.

The IXF18100 API module provides API calls to access the chip. This API includes the functions described in the following sections:

4.3.1.1 ixf18100Reset

Resets the chip, then reconfigures it.

Syntax

```
extern bb_Error_e
Ixf18100Reset(bb_ChipData_t *pChipData,
                   bb_ChipSegment_t *ptSegment,
                   bb_SelResetType_e resetType);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
ptSegment	bb_ChipSegment_t*	Resets chip section or block
ResetType	f18100_ResetType_e	Type of reset to perform: bb_RESET_GFP_TX_INTF bb_RESET_GFP_RX_INTF bb_RESET_GFP_CPU_INTF bb_RESET_GFP_TX_FCS bb_RESET_GFP_RX_FCS bb_RESET_GFP_RX_FSM bb_RESET_GFPbb_RESET_CHIP bb_RESET_LINE_INTF bb_RESET_SPI_INTF bb_RESET_APS_INTF

Returns

bb_Error_e	Error
b_NO_ERROR	Success

4.3.1.2 ixf18100InitChip

Initializes the chip based upon the configuration passed in by pChipData.

Note: The chip will be set offline (tri-stated and interrupts disabled) while initializing.

Syntax

```
extern bb_Error_e
Ixf18100InitChip(bb_ChipData_t *pChipData,
                      InitRegTable_t *pTable);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
pTable	InitRegTable_t*	Initializes data to be committed to the chip

Returns

bb_Error_e	Error
b_NO_ERROR	Success

4.3.1.3 ixf18100GetChipInfo

Gets the chip version and ID numbers.

Syntax

```
extern bb_Error_e
Ixf18100GetChipInfo (bb_ChipData_t *pChipData,
                     bb_ChipInfo_t *pChipInfo);
```

Input

pChipData	bb_ChipData_t*	Initializes chip data

Returns

bb_Error_e	Error
b_NO_ERROR	Success
pChipInfo ixf18100_ChipInfo_t	Chip information: ChipVersion ChipId

4.3.1.4 ixf18100InitAlarmCallback

Sets the pointer to the Alarm Callback Method. This is a user-defined function that can be called at the end of the `ixf18100_ChipIsr` routine allowing further processing of the collected alarm data.

Syntax

```
extern bb_Error_e
Ixf18100InitAlarmCallback(bb_ChipData_t *pChipData,
                          AlarmCallBack pAlarmCallbackArg);
```

Input

pChipData	bb_ChipData_t*	Initializes chip data
pAlarmCallback	AlarmCallBack	Points to an Alarm Callback function

Returns

bb_Error_e	Error
b_NO_ERROR	Success

4.3.1.5 ixf18100SetAlarmCfg

Sets the alarm configuration.

Syntax

```
extern bb_Error_e
Ixf18100SetAlarmCfg(bb_ChipData_t *pChipData,
                    bb_ChipSegment_t *section,
                    bb_AlarmType_e AlarmType,
                    void *pAlarmCfg);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
section	bb_ChipSegment_t*	Chip section or block
pAlarmCfg	bb_AlarmType_e	Alarm type to configure
pAlarmCfg	Void*	Alarm configuration data

Returns

bb_Error_e	Error
b_NO_ERROR	Success

4.3.1.6 ixf18100ChipIsr

This function is called to handle all interrupts that have been indicated by the chip. The interrupts are handled according to the hierarchy.

Syntax

```
extern bb_Error_e
Ixf18100ChipIsr(bb_ChipData_t *pChipData);
```

Input

pChipData	bb_ChipData_t*	Initializes chip data

Returns

bb_Error_e	Error
b_NO_ERROR	Success

4.3.1.7 ixf18100SetCfg

This function sets the configuration of POS watermarks, POS flow control, and the chip's GFC role.

Syntax

```
extern bb_Error_e
Ixf18100SetCfg(bb_ChipData_t *ptChipData,
                  bb_ChipSegment_t *ptSegment,
                  bb_SelConfig_e  SelCfg);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
ptSegment	bb_ChipSegment_t*	Block or section to configure
SelCfg	bb_SelConfig_e	Selects configuration type

Returns

bb_Error_e	Error
b_NO_ERROR	Success

4.3.1.8 ixf18100GetCfg

This function retrieves configuration informaiton.

Syntax

```
extern bb_Error_e
Ixf18100GetCfg(bb_ChipData_t *ptChipData,
                  bb_ChipSegment_t *ptSegment,
                  bb_SelConfig_e SelCfg);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
ptSegment	bb_ChipSegment_t*	Block or section to retrieve from
SelCfg	bb_SelConfig_e	Status type

Returns

bb_Error_e	Error
b_NO_ERROR	Success

4.3.1.9 ixf18100GetStatus

Retrieves the status of the Rx AIS, Input Clock Activity, and far-end GFC role.

Note: The test for the far-end GFC Role depends upon having active ATM traffic.

Syntax

```
extern bb_Error_e
Ixf18100GetStatus(bb_ChipData_t *pChipData,
                  bb_ChipSegment_t *section,
                  bb_SelStatus_e selStatus,
                  void *pStatus);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
section	bb_ChipSegment_t*	Block or section to retrieve status from
SelStatus	bb_SelStatus_e	The status type to retrieve

Returns

bb_Error_e	Error
b_NO_ERROR	Success
pStatusvoid*	Place to put Status

4.3.1.10 ixf18100GetCounters

Retrieves a set of Rx/Tx counters for a selected OHT type, ATM or POS, on a per-channel basis.

Syntax

```
extern bb_Error_e
Ixf18100GetCounters(bb_ChipData_t *ptChipData,
                    bb_ChipSegment_t *ptSection,
                    bb_SelCounters_e eCounter,
                    void *pCounters);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
ptSection	bb_ChipSegment_t*	Chip block or segment
eCounter	bb_SelCounters_e	The set of counters to retrieve

Returns

bb_Error_e	Error
b_NO_ERROR	Success
pCounters void*	A pointer to a Counter Structure that corresponds to the Selected Counters to retrieve.

4.3.1.11 ixf18100SetOpMode

Syntax

```
extern bb_Error_e
Ixf18100SetOpMode(bb_ChipData_t *pChipData,
                  bb_ChipSegment_t *section,
                  bb_OperMode_e opMode,
                  void *pModeCfg);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
section	bb_ChipSegment_t*	Chip block or segment to set
opMode	bb_OperMode_e*	Mode to set
pModeCfg	Void*	Configuration data to commit to device

Returns

bb_Error_e	Error
b_NO_ERROR	Success

4.3.1.12 ixf18100GetOpMode

Syntax

```
extern bb_Error_e
Ixf18100GetOpMode(bb_ChipData_t *pChipData,
                  bb_ChipSegment_t *section,
                  bb_OperMode_e *opMode,
                  void *pModeCfg);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
section	bb_ChipSegment_t*	Chip block or segment to set
opMode	bb_OperMode_e*	Mode to get
pModeCfg	Void*	Configuration data to retrieve

Returns

bb_Error_e	Error
b_NO_ERROR	Success

4.3.1.13 ixf18100CfgTest

Runs a specific test on the IX18100 chip.

Syntax

```
extern bb_Error_e
Ixf18100CfgTest(bb_ChipData_t *pChipData,
                bb_ChipSegment_t *ptSegment,
                bb_TestType_e testType,
```

```
void *pTestCfg);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
section	bb_ChipSegment_t*	Chip block or segment to test
testType	bb_TestType_e	Type of test

Returns

bb_Error_e	Error
b_NO_ERROR	Success
pTestCfg void*	A pointer to a Structure containing the test results.

4.3.1.14 ixf18100Read

Syntax

```
extern bb_Error_e
Ixf18100Read(bb_ChipData_t *pChipData,
              bb_Word_Size_t wordSize,
              ulong address,
              ushort length,
              void *buffer);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
wordSize	bb_Word_Size_t*	Enum size of data to be read: ONE_BYTE = 1 TWO_BYTES = 2 FOUR_BYTES = 4 EIGHT_BYTES = 8
address	ulong	Offset from chip base address to begin read
length	ushort	Number of words to read
buffer	Void*	Pointer to a structure in which to place the read results

Returns

bb_Error_e	Error
b_NO_ERROR	Success
buffer	Buffer contains read results

4.3.1.15 ixf18100Write

Syntax

```
extern bb_Error_e
Ixf18100Write(bb_ChipData_t *pChipData,
              bb_Word_Size_t wordSize,
              ulong address,
              ushort length,
              void *buffer);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
wordSize	bb_Word_Size_t*	Enum size of data to be read: ONE_BYTE = 1 TWO_BYTES = 2 FOUR_BYTES = 4 EIGHT_BYTES = 8
address	ulong	Offset from chip base address to begin write
length	ushort	Number of words to write
buffer	Void*	Pointer to a structure that contains the data to be written

Returns

bb_Error_e	Error
b_NO_ERROR	Success

4.3.1.16 ixf18100GetBuildVersion

Returns information specific to the driver build.

Syntax

```
extern bb_Error_e
Ixf18100GetBuildVersion(bb_ChipData_t *pChipData,
                        char *drvName,
                        char *date,
                        ushort *buildVer,
                        ushort *buildRev);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
drvName	Char*	Character pointer to a buffer for the driver name
date	Char*	Character pointer to a buffer for the driver date
buildVer	ushort	Variable for build version
buildRev	ushort	Variable for build revision

Returns

bb_Error_e	Error
b_NO_ERROR	Success
drvName Char*	Character pointer to a buffer containing the driver name
date Char*	Character pointer to a buffer containing the driver date
buildVer ushort	Variable containing build version
buildRev ushort	Variable containing build revision

4.3.1.17 ixf18100InitAllocMemory

Allocates memory to support the driver data structures.

Syntax

```
extern bb_Error_e
Ixf18100InitAllocMemory(bb_ChipData_t *pChipData);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data

Returns

bb_Error_e	Error
b_NO_ERROR	Success

4.3.1.18 ixf18100DeAllocMemory

Deallocates the memory used to support the driver data structures.

Syntax

```
extern bb_Error_e
Ixf18100DeAllocMemory(bb_ChipData_t *pChipData);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data

Returns

bb_Error_e	Error
b_NO_ERROR	Success

4.3.1.19 ixf18100XgmacGetAddress

Syntax

```
extern bb_Error_e
Ixf18100XgmacGetAddress(bb_ChipData_t *pChipData,
                        bb_ChipSegment_t *section,
                        IxfApi_MacAddress_t *pMacAddress);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
section	bb_ChipSegment_t*	Chip block or section
address	Void*	Pointer to a buffer for the 48-bit MAC address to be read

Returns

Type	Description
bb_Error_e	Error
b_NO_ERROR	Success
address void*	Pointer to a buffer that contains the 48-bit MAC address read

4.3.1.20 ixf18100XgmacSetAddress

Function Definition

```
extern bb_Error_e
Ixf18100XgmacSetAddress(bb_ChipData_t *pChipData,
                        bb_ChipSegment_t *section,
```

```
IxfApi_MacAddress_t *pMacAddress);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
ptSection	bb_ChipSegment_t*	Chip block or section
address	Void*	Pointer to a buffer for the 48-bit MAC address to be written

Returns

bb_Error_e	Error
b_NO_ERROR	Success

4.3.1.21 ixf18100SonetGetWindowSize

Gets the window size for the degraded and excessive error.

Syntax

```
extern bb_Error_e
Ixf18100SonetGetWindowSize(bb_ChipData_t *pChipData,
                           bb_ChipSegment_t *section,
                           bb_WindowSizeMode_e mode,
                           ulong *value);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
ptSection	bb_ChipSegment_t*	Chip block or section
mode	bb_WindowSizeMode_e	Parameter to identify the kind of error (degraded or excessive) if the window is being cleared or set

Returns

bb_Error_e	Error
b_NO_ERROR	Success
value ulong*	Parameter to store the value of the window size

4.3.1.22 ixf18100SonetSetWindowSize

Sets the window size for the degraded and excessive error.

Syntax

```
extern bb_Error_e
Ixf18100SonetSetWindowSize(bb_ChipData_t *pChipData,
                           bb_ChipSegment_t *section,
                           bb_WindowSizeMode_e mode,
                           ulong value);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
ptSection	bb_ChipSegment_t*	Chip block or section
mode	bb_WindowSizeMode_e	Parameter to identify the kind of error (degraded or excessive) if the window is being cleared or set
value	ulong	Value of the window size

Returns

bb_Error_e	Error
b_NO_ERROR	Success

4.3.1.23 ixf18100SonetGetTrace

Get the expected,received ortransmitted J0/J1 Path Trace. The trace string will be returned in the format set in the corresponding Configuration Format bits.

Syntax

```
extern bb_Error_e
Ixf18100SonetGetTrace(bb_ChipData_t *ptChipData,
                      bb_ChipSegment_t *ptSection,
                      bb_TraceType_e TraceType,
                      char *pTrace,
                      ushort *pLength);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
ChanNum	bb_ChipSegment_t*	Chip block or segment
TraceType	bb_TraceType_e	Type of Trace: bb_EXPECTED_J0 bb_EXPECTED_J1 bb_TX_J0 bb_TX_J1 bb_RX_J0 bb_RX_J1

Returns

bb_Error_e	Error
b_NO_ERROR	Success
pTrace Char*	Place to return null-terminated Rx Trace String Note: at least 65 bytes long
pLength Ushort*	

4.3.1.24 ixf18100SonetSetTrace

Sets the expected and the Transmit J0/J1 Trace.

Syntax

```
extern bb_Error_e
Ixf18100SonetSetTrace(bb_ChipData_t *ptChipData,
                      bb_ChipSegment_t *ptSection,
                      bb_TraceType_e TraceType,
                      bb_TraceFormat_e TraceFormat,
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
ptSection	bb_ChipSegment_t*	Chip block or segment
TraceType	bb_TraceType_e	Type of Trace: bb_EXPECTED_J0 bb_EXPECTED_J1 bb_TX_J0 bb_TX_J1 bb_RX_J0 bb_RX_J1
TraceFormat	bb_TraceFormat_e	The format of the Trace String defined as: bb_64_BYTE_WITH_LF_CR bb_64_BYTE_FREE_FORM bb_16_BYTE_WITH_CRC7 bb_1_BYTE bb_IGNORE_RX_TRACE bb_DEFAULT_TX_TRACE
pTrace	uchar*	The null-terminated Trace String. Note: This Method will format the pTrace string to comply with the selected TraceFormat (i.e.: Driver adds CRC7, or LF/CR, if needed)

Returns

bb_Error_e	Error
b_NO_ERROR	Success

4.3.1.25 ixf18100SonetGetOhBytes

This funtion will retrieve the specified Overhead Bytes: K1, K2, Expected/Tx C2, HPT RDI in G1.

Syntax

```
extern bb_Error_e
Ixf18100SonetGetOhBytes(bb_ChipData_t *ptChipData,
                        bb_ChipSegment_t *ptSection,
                        bb_SelOhBytes_e  SelOhBytes,
                        void *pOhBytes);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
ptSection	bb_ChipSegment_t*	Chip block or section
SelOhByte	bb_SelOhByte_e	Specific overhead bytes to get: bb_RX_MST_BYTES, bb_RX_HPT_BYTES

Returns

bb_Error_e	Error
b_NO_ERROR	Success
pOhBytes void*	Place to put Overhead Bytes, of type: bb_RxMstBytes_t*, or ixf18100_RxHptBytes_t*

4.3.1.26 ixf18100SonetSetOhBytes

Sets an Overhead Byte: K1, K2, Expected/Tx C2, HPT RDI in G1.

Syntax

```
extern bb_Error_e
Ixf18100SonetSetOhBytes(bb_ChipData_t *ptChipData,
                        bb_ChipSegment_t *ptSection,
                        bb_SelOhBytes_e SelOhByte,
                        ushort OhByte);
```

Input

pChipData	bb_ChipData_t*I	Initializes chip data
ptSection	bb_ChipSegment_t*	Chip block or section
SelOhByte	bb_SelOhByte_e	Specific overhead bytes to get: bb_RX_MST_BYTES, bb_RX_HPT_BYTES
OhByte	ushort	Value of the selected overhead byte

Returns

bb_Error_e	Error
b_NO_ERROR	Success

4.4 The IXF18100 Device Specific Driver

4.4.1 Common Data Structure

The driver will pass the common structure to all the functions. The common structure data members are listed in the following structure.

```
typedef struct  /* Complete Data for a Chip */
{
   bb_RegPointer_type   BaseAddress; /* Base Address of chip */
   bb_ChipType_e        ChipType;      /* Type of Chip */
   void*                pChipCfg; /* Pointer to Chip Specific Configuration */
   void*               pAlarmCfg; /* Pointer to Chip Specific Alarm Config */
   void*                funcPtr;       /* Pointer to the chip's api func's */
} bb_ChipData_t;


typedef struct
{
    ixf18100_ChipLevelCfg_t   ChipLevelCfg;   /* Chip configuration */
    ixf18100_Spi4Cfg_t        Spi4Cfg;        /* SPI-4 configuration */
#ifndef INCLUDE_18104_LIB
    ixf18100_SonetCfg_t       SonetCfg;       /* Sonet configuration */
    ixf18100_PosCfg_t         PosCfg;         /* POS configuration */
    ixf18100_GfpCfg_t         GfpCfg;         /* GFP configuration */
#endif
#ifndef INCLUDE_18102_LIB
    ixf18100_PcsCfg_t         PcsCfg;         /* PCS configuration */
    ixf18100_XgmacCfg_t       XgmacCfg;       /* XGMAC configuration */
#endif
} ixf18100_ChipCfg_t;
```

This structure ixf18100_ChipCfg_t is in file ixf18100_cnfg_d.h . For a complete list of ixf18100 data structures please refer ixf18100d.h file.

4.4.2 Error Codes

The following tables contain a complete list of error codes returned by the driver.

Error Enumerator

```
typedef enum
{
  bb_NO_ERROR = 0,              /* Returned by Driver for no error */
```

Table 4-2. Fatal Error Types and Descriptions

Error	Description
bb_FATAL_ERROR	Fatal error codes should be defined here
bb_GENERAL_ERROR	Catch all type of error
bb_NULL_ADDRESS_ASSIGNED	A NULL Base Address has been assigned

Table 4-3. Common Error Types and Descriptions,,

Error	Description
bb_NO_CHIP_DATA = bb_COMMON_ERROR_OFFSET	pChipData = 0
bb_NULL_BASE_ADDR	BaseAddress = 0, for chip
bb_INV_BASE_ADDR	BaseAddress not for initialized chip
bb_INV_CHIP_TYPE	Chip type not supported
bb_NO_CHIP_CFG	pChipCfg = 0, for chip
bb_NO_ALARM_CFG	pAlarmCfg = 0, for chip, but Alarm cfg needed
bb_UNDEF_ALARM_BITS	An XxxAlarmCfg uses undefined alarm bits
bb_STM_MODE_MISMATCH	Mismatch between h/w and s/w cfg of STM-0/1
bb_INV_SEL_OH_BYTE	Invalid SelOhByte
bb_INV_SEL_COUNTERS	Invalid SelCounters
bb_INV_CHAN_TEST	Invalid Test Channel
bb_INV_PARAMETER	Invalid parameter, generic error
bb_INV_CHIP_SEGMENT	Chip segment is invalid
bb_NULL_ARG	null pointer passed as argument to function
bb_INV_BLOCK_OPERATION	operation not supported on this block
bb_FN_NOT_SUPPORTED	function not supported

Table 4-4. OHT Error Types and Descriptions ,

Error	Description
bb_JN_TRACE_WRITE_FAIL = bb_OHT_ERROR_OFFSET	Write of Expected/Rx/Tx J0,J1,J2 Trace failed
bb_INV_EXP_JN_FMT	The Expected Jn Format is invalid
bb_TX_J1_FOR_RPTR	Cannot set Tx J1 trace for repeater
bb_POH_PASSTHRU	Illegal call; all POH bytes passed through
bb_OHT_NOT_IN_TEST	OHT must be in test mode, to introduce errors
bb_TX_J0_NOT_CPU	For Terminal or ADM, Tx J0 source must = CPU
bb_INV_EXP_J1_FMT	An Invalid Expected J1 Format found
bb_INV_TX_J1_FMT	An Invalid Tx J1 Format found
bb_NO_OHT_NU_CFG	pNuBytes = 0
bb_NOT_PROTECTING_MAIN	not Main Terminal or ADM, or no Protection h/w

Error	Description
bb_NOT_TERM_ADM	This function valid only for Terminal or ADM
bb_INV_BKUP_OHT	Invalid Protection Allocation
bb_INV_SEL_OHT_CFG	Invalid SelOhtCfg Value Used
bb_TRACE_ACCESS_FAIL	Trace read or write has failed
bb_INV_TRACE_FORMAT	Invalid trace format selected
bb_INV_TRACE_TYPE	Invalid trace type selected

Table 4-5. Mapper Error Types and Descriptions,

Error	Description
bb_INV_PORT_NUM = bb_MAPPER_ERROR_OFFSET	Invalid PortNum
bb_NO_MPR_CFG	pMprCfg = 0
bb_INV_ALM_SPEC	Invalid AlmSpec
bb_NO_ADM_CFG	pAdmCfg = 0
bb_NOT_ADM	This chip not configured as an ADM
bb_INV_TIMESLOT	ADM timeslots = 0, or 1-28
bb_GET_J2_TYP_ERR	
bb_SET_J2_TYP_ERR	
bb_SET_J2_LEN_ERR,	
bb_PORT_NOT_N2_ENBLD	
bb_INV_SIG_LBL	
bb_INV_SEL_AUTO_FDBK	Invalid Auto Feedback spec
bb_INV_TEST_CNFG	
bb_Error_e	

4.5 VxWorks and Linux Ixf18100 Device Driver

The IXF API device driver for the IXF18100 device is the same whether it runs under VxWorks or Linux with a few exceptions. These exceptions are due to #ifdef IXF_LINUX OS conditional statements and the addition of a Linux OS related support files. The software is implemented in standard ANSI C, which allows the code to be highly portable and fully reentrant.

The VxWorks kernel version under which this driver will function is 5.5 and the Linux kernel version is 2.4. The API routines listed below are applicable for both VxWorks and Linux. This guide should be used as a companion to the IXF API User's Guide which explains the APIs in more detail. The driver runs as an application in VxWorks and as a kernel module in Linux. The kernel module for Linux does not implement any communication to user mode. It is only used by other code that operates in kernel mode.

The driver exports an interrupt handling routine (IxfApiChipIsr) to handle alarms. See the IxfApiChipIsr section to learn how to handle the low level interrupts from the processor and call the IxfApiChipIsr to handle the alarms within the device.

Most of the API routines accept a handle describing the type of device, where the device is located, and some configuration structures. Many API routines expect a "section" parameter, this tells the routine the location of the item the user wants to set or retrieve. An example of what should be populated in the section (of type bb_ChipSegment_t) is the channel number. Another expected parameter in many of the API routines will be an enumerated type applicable to the functionality of the routine. Again, the IXF API User's Guide should be referenced for more information. The following API descriptions attempt to highlight the possible values for each routine as well as alternative device specific routines the user can use.

IxfApiInitChip

This routine sends an internal RAM-based initialization table of register values to the device. The table is identified in the structure bb_Chip_Data_t, a pointer to which is passed to this function as an input parameter. The valid value for the base address is the proper physical or virtual address of the device in the system and the chip type should be set to bb_18101_CHIP. Under Linux, the calling code is expected to pass the handle returned from ioremap() for the correct virtual address.

IxfApiAllocDataStructureMem

This routine dynamically allocates memory within the pChip_Data parameter for members that need dynamic allocation. The user calls this routine to enable alarm capabilities and chip initialization. This routine must be called before IxfApiGetCfg, IxfApiSetCfg, and IxfApiSetAlarmCfg are used. The pChipData pointer must be initialized via IxfApiInit prior to calling this function.

IxfApiDeAllocMemory

This routine frees all memory allocated during the IxfApiAllocDataStructureMem call.

IxfApiSetAlarmCfg

This routine is used to modify interrupt enable masks for a specific set of interrupts.

IxfApiReset

This routine resets all or some portion of the device. The alternative device specific driver would be Ixf18100Reset. The valid values for the ResetType enumeration parameter are:

Resets the entire chip:

```
   bb_RESET_CHIP
```

Resets portions of the GFP interface:

```
bb_RESET_GFP_TX_INTF
bb_RESET_GFP_RX_INTF
bb_RESET_GFP_CPU_INTF
bb_RESET_GFP_TX_FCS
bb_RESET_GFP_RX_FCS
bb_RESET_GFP_RX_FSM
bb_RESET_GFP
```

Resets only the line interface:

```
bb_RESET_LINE_INTF.
```

Resets the SPI4 interface:

```
bb_RESET_SPI_INTF.
```

Resets the APS interface:

```
bb_RESET_APS_INTF
```

IxfApiGetChipInfo

This routine returns the chip related information. This routine retrieves a pointer to the bb_ChipInfo_t containing the chip id and chip version numbers. The alternative to this routine for the device specific driver would be Ixf18100GetChipInfo.

```
typedef struct              /* Information set by the hardware */
{
   uchar ChipVersion;       /* Chip Version */
   ushort  ChipId;          /* Chip ID */

} bb_ChipInfo_t;
```

Alternative for the device specific driver:

```
typedef struct              /* Information set by the hardware */
{
   ushort ChipVersion;      /* Chip Version */
   ushort ChipId;           /* Chip ID */
} ixf18100_ChipInfo_t;
```

IxfApiGetTrace

This routine retrieves a SONET J0/J1 trace. These traces are embedded in the SONET overhead frames and this routine is used to extract the stream of bytes from device. The valid values for the type of trace the user can retrieve are:

- bb_EXPECTED_J0
- bb_EXPECTED_J1
- bb_RX_J0
- bb_RX_J1
- bb_TX_J0
- bb_TX_J1

IxfApiSetTrace

- This routine sets a SONET J0/J1 trace. The valid values for the type of trace the user can set are: bb_EXPECTED_J0
- bb_EXPECTED_J1
- bb_RX_J0
- bb_RX_J1

- bb_TX_J0
- bb_TX_J1.

The valid values for the type of trace format are:

- bb_64_BYTE_WITH_LF_CR
- bb_64_BYTE_FREE_FORM
- bb_16_BYTE_WITH_CRC7
- bb_IGNORE_RX_TRACE
- bb_1_BYTE

IxfApiSetCfg

This routine is used to configure the chip. Many types of configurations can be performed with the combinations of SelCfg. The configuration is passed within the pChipData variable. Valid enum values for sections within the ifx18100 chip are:

- ixf_eSPI4
- ixf_epos
- ixf_eGFP
- ixf_ePCS
- ixf_eXGMAC
- ixf_eAPS

IxfApiGetCfg

This routine is used to get the configuration of the chip. Many types of configurations can be performed with the combination of SelCfg. The chip configuration is returned within the pChipData variable. The same valid enum values for sections used in IxfApiSetCfg apply to this function as well.

IxfApiGetStatus

This routine returns status information. The status returned is comprised of register or register bit values that provide various status information from the device.

IxfApiGetCounters

This routine retrieves counter values. This can be in the form of a single counter or a group of counters. The alternative device specific routine is Ixf18100GetCounters. The enums for valid sections or blocks in the device where counter(s) may be retrieved are:

- ixf_eSONET
- ixf_eSPI4
- ixf_ePOS
- ixf_eGFP

- ixf_ePCS
- ixf_eAPS
- ixf_eXGMAC

The valid individual counter request values are:

- ixf_eSONET:

```
bb_OOF_CNT                          /* Out of Frame Event Counter*/
bb_B1_BIP_ERR_CNT                   /* B1 Bip errors */
bb_B1_BLOCK_ERR_CNT                 /* B1 Block errors */
bb_B2_BIP_ERR_CNT                   /* B2 Bip errors */
bb_B2_BLOCK_ERR_CNT                 /* B2 Block errors */
bb_MST_REI_BIP_ERR_CNT              /* REI Bit errors */
bb_MST_REI_BLOCK_ERR_CNT            /* REI Block errors */
bb_IN_AU_NCNT     /* Count of Incoming Negative AU Ptr Justifications */
bb_IN_AU_PCNT     /* Count of Incoming Positive AU Ptr Justifications */
bb_SONET_COUNTERS                   /* All SONET counters. */
```

- ixf_eSPI4:

```
bb_SPI4_RX_BUS_ERR_CNT
bb_SPI4_RX_FULL_ERR_CNT
bb_SPI4_RX_DIP2_ERR_CNT
bb_SPI4_RX_NO_EOP_ERR_CNT
bb_SPI4_RX_COUNTERS
bb_SPI4_TX_BUS_ERR_CNT
bb_SPI4_TX_PAR_ERR_CNT
bb_SPI4_TX_FULL_ERR_CNT
bb_SPI4_TX_NO_EOP_ERR_CNT
bb_SPI4_TX_COUNTERS
bb_SPI4_COUNTERS/* All SPI4 counters. */
```

- ixf_epos:

```
bb_POS_RX_GOOD_FRM_CNT
bb_POS_RX_GOOD_BYTE_CNT
bb_POS_RX_ABORTED_FRM_CNT
bb_POS_RX_ABORTED_BYTE_CNT
bb_POS_RX_FCS_ERR_FRM_CNT
bb_POS_RX_FCS_ERR_BYTE_CNT
bb_POS_RX_MIN_PLE_CNT
bb_POS_RX_MAX_PLE_CNT
bb_POS_RX_COUNTERS
bb_POS_TX_FRM_CNT
bb_POS_TX_BYTE_CNT
bb_POS_TX_COUNTERS
bb_POS_COUNTERS                     /* All Counters for POS */
```

- ixf_eGFP:

```
bb_GFP_RX_FRM_SBC_HEC_FAIL_CNT
bb_GFP_RX_FRM_MBC_HEC_CNT
bb_GFP_RX_FRM_SBT_HEC_FAIL_CNT
bb_GFP_RX_FRM_MBT_HEC_CNT
bb_GFP_RX_FRM_EHEC_CNT
bb_GFP_RX_CTRL_FRM_CRC_ERR_CNT
```

```
bb_GFP_RX_FRM_FCS_ERR_CNT
bb_GFP_RX_CTRL_FRM_CNT
bb_GFP_RX_LARGE_FRM_CNT
bb_GFP_RX_GOOD_FRM_CNT
bb_GFP_RX_CTRL_INT_CNT
bb_GFP_RX_THEC_ERR_INT_CNT
bb_GFP_RX_EHEC_ERR_INT_CNT
bb_GFP_RX_FCS_ERR_INT_CNT
bb_GFP_RX_COUNTERS
bb_GFP_TX_GOOD_FRM_CNT
bb_GFP_TX_ERR_FRM_CNT
bb_GFP_TX_COUNTERS
bb_GFP_COUNTERS/* All GFP counters. */
```

- ixf_ePCS:

```
bb_PCS_BER_CNT
bb_PCS_ERR_BLK_CNT
bb_PCS_JITTER_ERR_CNT
bb_PCS_COUNTERS/* All PCS counters. */
```

- ixf_eAPS:

```
bb_OOF_CNT                    /* Out of Frame Event Counter*/
bb_B1_BIP_ERR_CNT             /* B1 Bip errors */
bb_B1_BLOCK_ERR_CNT           /* B1 Block errors */
bb_B2_BIP_ERR_CNT             /* B2 Bip errors */
bb_B2_BLOCK_ERR_CNT           /* B2 Block errors */
bb_MST_REI_BIP_ERR_CNT        /* REI Bit errors */
bb_MST_REI_BLOCK_ERR_CNT      /* REI Block errors */
bb_SONET_COUNTERS             /* All SONET counters. */
```

- ixf_eXGMAC:

```
bb_XGMAC_TX_OK_OCTS_CNT
bb_XGMAC_TX_OK_MLTCAST_FRM_CNT
bb_XGMAC_TX_OK_BRDCAST_FRM_CNT
bb_XGMAC_TX_OK_FRM_CNT
bb_XGMAC_TX_64_OCT_FRM_CNT
bb_XGMAC_TX_65_TO_127_OCT_FRM_CNT
bb_XGMAC_TX_128_TO_255_FRM_CNT
bb_XGMAC_TX_256_TO_511_FRM_CNT
bb_XGMAC_TX_OK_512_TO_1023_CNT
bb_XGMAC_TX_OK_1024_TO_15XX_CNT
bb_XGMAC_TX_OK_15XX_TO_MAX_CNT
bb_XGMAC_TX_VLAN_FRM_CNT
bb_XGMAC_TX_PAUSE_CTRL_FRM_CNT
bb_XGMAC_TX_UNICAST_FRM_CNT
bb_XGMAC_TX_MAC_CTRL_FRM_CNT

bb_XGMAC_RX_OK_OCTS_CNT
bb_XGMAC_RX_OK_MLTCAST_FRM_CNT
bb_XGMAC_RX_OK_BRDCAST_FRM_CNT
bb_XGMAC_RX_OK_FRM_CNT
bb_XGMAC_RX_64_OCT_FRM_CNT
bb_XGMAC_RX_65_TO_127_OCT_FRM_CNT
bb_XGMAC_RX_128_TO_255_FRM_CNT
```

```
bb_XGMAC_RX_256_TO_511_FRM_CNT
bb_XGMAC_RX_OK_512_TO_1023_CNT
bb_XGMAC_RX_OK_1024_TO_15XX_CNT
bb_XGMAC_RX_OK_15XX_TO_MAX_CNT
bb_XGMAC_RX_VLAN_FRM_CNT
bb_XGMAC_RX_PAUSE_CTRL_FRM_CNT
bb_XGMAC_RX_UNICAST_FRM_CNT
bb_XGMAC_RX_MAC_CTRL_FRM_CNT
bb_XGMAC_RX_ETHER_STATS_USIZE_PKTS_CNT
bb_XGMAC_RX_ETHER_STATS_OSIZE_PKTS_CNT
bb_XGMAC_RX_ETHER_STATS_OCTS_CNT
bb_XGMAC_RX_ETHER_STATS_PKTS_CNT
bb_XGMAC_RX_ETHER_STATS_FRGMNTS_CNT
bb_XGMAC_RX_ETHER_STATS_JABBERS_CNT
bb_XGMAC_RX_FRM_CHK_SEQ_ERR_CNT
bb_XGMAC_TX_COUNTERS
bb_XGMAC_RX_COUNTERS
bb_XGMAC_COUNTERS/* All XGMAC counters. */
```

IxfApiSetOhBytes

This routine sets Overhead bytes. The alternative routine is the Ixf18100SetOhBytes API. The valid overhead bytes are:

- ixf_eSONET:

```
bb_K1
bb_TX_K1
bb_K2
bb_TX_K2
bb_K1_K2
bb_K3
bb_TX_K3
bb_S1
bb_TX_S1
bb_EXPECTED_C2
bb_RECEIVED_C2
bb_TX_C2
bb_G1
bb_TX_HPT_RDI_IN_G1
bb_K3_Z4
bb_M1
bb_M0
bb_MNU
bb_RNU1
bb_RNU2
bb_RNU9
bb_J1
bb_B3
bb_F2
bb_H4
bb_Z3_F3
bb_Z4_K3
```

```
bb_Z5_N1
bb_LINE_OH_BYTES
bb_PATH_OH_BYTES
```

- ixf_eAPS:

```
bb_K1
bb_K2
bb_LINE_OH_BYTES
bb_PATH_OH_BYTES
```

IxfApiGetOhBytes

This routine retrieves overhead bytes. The alternative routine is the Ixf18100GetOhBytes API. The valid overhead bytes are:

- ixf_eSONET:

Same as in IxfSetOhBytes.

- ixf_eAPS:

```
bb_K1
bb_K2
bb_S1
bb_LINE_OH_BYTES
bb_PATH_OH_BYTES
```

IxfApiSetOpMode

This routine is used to set the operating mode of the chip. Though the driver and IXF181001 support additional modes, the IXD28192 board is limited to the following valid enum values:

```
ETHERNET_MODE
WAN_PHY_MODE
POS_MODE
```

IxfApiGetOpMode

This routine is used to get the operating mode of the chip. It is the compliment of IxfApiSetOpMode.

IxfApiCfgTest

This routine is used to configure the chip in the specified test mode. The valid enum values for testType in the ixf1810x chip are:

```
bb_SYSTEM_LOCAL_LOOPBACK
bb_SYSTEM_REMOTE_LOOPBACK
bb_SPI4_TX_LOCK_DIS
bb_SPI4_TX_FIFO_STATUS
bb_LINE_LOCAL_LOOPBACK
bb_LINE_REMOTE_LOOPBACK
```

IxfApiGenericRead

This routine will read data from the device from the offset specified. The routine is passed a pointer to a structure of type bbChipData to identify the chip base address and type. Additionally, wordSize identifies the number of bytes in a word, address is the offset from the base address, and length indicates the number of words to read. A void pointer to buffer into which the read data is placed is also passed to the routine.

IxfApiGenericWrite

This routine will write data to the device from the offset specified. The parameters are the same as the IxfApiGenericRead with the exception that buffer contains data to be written to the indicated address.

IxfApiInitAlarmCallback

This routine registers a callback with the driver. The callback will be called whenever any alarm occurs in the system. The argument for this routine is a function pointer pointing to the callback function.

IxfApiChipIsr

This routine handles alarms. Its only parameter is the handle.

IxfApiGetWindow

This routine is called to retrieve window related registers.

IxfApiSetWindow

This routine is called to set window related registers.

IxfApiGetMacAddress

This routine retrieves the 48-bit MAC address.

IxfApiSetMacAddress

This routine sets the 48-bit MAC address.

4.6 Utilities/Tools

4.6.1 Intel Optical Component Management Software (OCMS)

The OCMS is a Graphical User Interface, which allows the user to easily configure the device via a TCP/IP Sockets communication link. OCMS uses Ethernet as its physical layer protocol medium. In order to configure the device, the user must have the knowledge of the device in order to select the proper configuration options or the user must have a working configuration file.

More details will be added in the future to this section on the OCMS. As this will not delay driver development, the intent is to release this document in it's current form.

- Proprietary Protocol System

  The protocol is a proprietary protocol that allows the user interface to communicate to the embedded target. This protocol encodes opcodes on the host side and decodes these same opcodes on the target side. The protocol depends on the Communication Layer to provide the means of transporting the data. The protocol depends on the IXF API layer and the OCMS to carry out the request as specified by the received opcode.

- Communication Layer

  This layer provides the standard BSD Sockets support. For VxWorks implementation, the server will be listening on port 700. In the Linux implementation, the server will be listening on port 5030.

4.7 The IXDP28192 Driver Unit Tests

These tests will be conducted only on the IXDP28192 daughter card independent of rest of the system. These tests will be considered complete if all tests yield a "PASSED" result. For further test coverage of the media card, the user should consult the Deer Island Diagnostic LLD.

- IxfApiInit Test
  - §Configure IXFAPI driver for the IXF18100 device.
  - §Tests return status of the routine.
- IxfApiInitChip Test
  - §Initializes the device with a certain configuration.
  - §Tests return status of the routine.
- IxfApiAllocDataStructureMem Test
  - §Configure IXFAPI driver to allocate any necessary memory needed by the device driver.
  - §Tests return status of the routine.
- IxfApiDeAllocMemory Test
  - §Configure IXFAPI driver to deallocate any memory allocated during the IxfApiAllocDataStructureMem.
  - §Tests return status of the routine.
- IxfApiGetCounters Test
  - §Retrieves a certain counter value from the device.
  - §Tests return status of the routine.
- IxfApiReset Test
  - §This test writes a certain configuration to the device using IxfApiInitChip then uses this Reset routine to reset the entire device. Once this is done, a comparison of the known default values is done.
  - §Tests return status of the routine.
- 5.1.7IxfApiGetChipInfo Test

- — §Retrieves the chip id and chip version from the device and compares the value to what is expected.
- — §Tests return status of the routine.

- IxfApiGetBuildVersion Test
  - — §Retrieves the build version from the driver and compares the value to what is expected.
  - — §Tests return status of the routine.
- IxfApiGenericRead Test
  - — §Retrieves the current value in the same location as what the GenericWrite wrote to and compare the value
  - — §Tests return status of the routine.
- IxfApiGenericWrite Test
  - — §Writes a certain value to a certain location then the GenericRead will compare.
  - — § Tests return status of the routine.
- IxfApiGetTrace Test

  §Gets a particular SONET J0/J1 Trace String at the same location in the SetTrace test, then compares the result to what was expected.

  §Tests return status of the routine.
- IxfApiSetTrace Test
  - — §Sets a particular SONET J0/J1 Trace String at a certain location. This test is used with the GetTrace routine.
  - — §Tests return status of the routine.
- IxfApiGetWindowSize Test
  - — §Retrieves the current value in the window size register and compares the value to what was written in SetWindowSize.
  - — §Tests return status of the routine.
- IxfApiSetWindowSize Test
  - — §Writes a window size value and use the IxfApiGetWindowSize to compare what was written.
  - — §Tests return status of the routine.
- IxfApiGetOhBytes Test
  - — §Gets a particular SONET overhead byte at the same location in the SetOhByte test, then compares the result to what was expected.
  - — §Tests return status of the routine.
- IxfApiSetOhBytes Test
  - — §Sets a particular SONET overhead byte at a certain location. This test is used with the GetOhByte routine.
  - — § Tests return status of the routine.

Single OC-48, Quad OC-12 I/O Card 5

The documents “Intel® IXF API User Guide” and “Intel® IXF6048 API User Guide” describe the relevant device driver API and is published on Field Division Business Link (FDBL). Please contact your Intel representative for access.