Ethernet Device Driver Software Interface

for Myricom Myri-10G Network Interface Cards

This document is intended to help programmers who want to "port" the Myri10GE driver to other operating systems, or customize this driver. An understanding of the information contained in this reference document is not necessary for the installation and ordinary use of the Myri10GE driver.

For additional information, please refer to www.myri.com/.
For tech support, please contact help@myri.com.

Terms and Acronyms

DMA	Direct Memory Access.
DMA address	Address used by the NIC to access main memory through PCI Express.
INTx	Legacy PCI interrupt emulation.
IP	Internet Protocol.
LSB	Least Significant Byte.
LSW	Least Significant Word.
MAC	Media Access Control.
MSB	Most Significant Byte.
MSI	Message Signaled Interrupts.
MSS	Maximum Segment Size.
MSW	Most Significant Word.
MTU	Maximum Transmission Unit.
MXGEFW_SEND_SMALL_SIZE	1520 bytes.
NIC	Network Interface Card.
Non-TSO packet	TCP packet with payload of at most MSS bytes that does not require TSO.
PCIe	PCI Express.
PCIe address	DMA address.
PIO	Programmed I/O.
RDMA	Read DMA.
TCP	Transmission Control Protocol.
TSO	TCP Segmentation Offload.
TSO packet	TCP packet with payload larger than MSS.
TSO-segmented packet	Actual TCP packet sent by the NIC that contains a part of the payload in the TSO packet.
TSO-segmented payload	Payload of a TSO-segmented packet.
UDP	User Datagram Protocol.
VLAN	Virtual Local Area Network.
WDMA	Write DMA.
Word	32 bits (4 bytes).

1. Introduction

This document defines the software interface between the Myricom Myri-10G NICs and the Ethernet device driver. Myricom currently supports device drivers for a number of operating systems including Linux, Solaris, Windows, FreeBSD, and Mac OS X. This document is intended for device driver developers who wish to modify the existing driver provided by Myricom or wish to write a new driver for unsupported operating systems. NIC firmware developers may also use this document to understand the software interface between the NIC and the device driver.

1.1. Overview of the NIC Hardware

The NIC includes the Myricom Lanai-Z8E chip, which has an 8-lane (x8) PCI Express interface to a host computer and a 10-Gigabit Ethernet MAC. The Lanai chip includes a 32-bit programmable processor that runs firmware. The Lanai chip also includes on-chip buffers that are used to accelerate packet data transfers between the PCI Express interface and the 10-Gigabit Ethernet MAC. The Lanai accesses 2MB of fast SRAM on the NIC. Overall, the NIC has enough bandwidth to saturate the 10-Gigabit Ethernet link full-duplex.

The on-board EEPROM stores bootstrap firmware as well as other crucial information. Upon a system reset, the NIC runs the firmware stored in the EEPROM.

The NIC also implements a JTAG-based debugging mechanism. Contact Myricom for further information.

1.2. Communication with the NIC

All communications between the NIC and the device driver take a form of request and response. The device driver issues various requests such as requests to send packets by writing them to the NIC memory through PIO. Once the commands are processed by the NIC, the NIC generally transfers completion status of the requests to main memory, which is then processed by the device driver.

There are two categories of requests. First, the NIC provides the device driver with various commands that can be used to configure the NIC. These commands are normally used during the driver initialization. The driver issues only one request at a time and typically blocks until the request is processed by the NIC.

Second, there are requests for sending and receiving packets. These use rings of descriptors and are processed asynchronously by the NIC and the device driver in order to maximize performance.

1.3. Summary of Key Features

Ethernet flow control	The NIC supports hardware Ethernet flow control.
Multicast filtering	The NIC can filter received multicast packets based on their MAC addresses.
Jumbo packets	The NIC can transmit and receive packets of size up to 9400B.
Checksum offload	The NIC computes TCP/UDP checksums for transmit packets and computes Internet checksums for receive packets.
TCP segmentation offload (TSO)	The NIC fragments a large TCP packet into smaller TCP packets.
INTx and MSI interrupt modes	The NIC supports both legacy INTx emulation and MSI interrupt modes.
Firmware download	The driver can download firmware on the fly to upgrade the functionality of the NIC.

1.4. Endian

The NIC uses a 32-bit, big endian processor. Data structures throughout this document assume big endian. The device driver must ensure that all control data such as various descriptors and commands are in big endian. For example, on an Intel machine that uses little endian, in order to pass a 4B integer to the NIC, the device driver must byte swap the integer before handing it to the NIC.

1.5. PCI Memory Space

The NIC maps 16MB of memory space by default. The device driver uses this memory space for all PIO accesses. The layout of the memory space is shown in Table 1. The NIC also maps another memory space of size 1MB. This memory space is not used for normal operations.

Region Name	Offset (B)	Size	Description
SRAM	0x0	2MB	On-board 2MB SRAM.
ETH_SEND_4	0x200000	64B	Device driver writes 4 new Send Descriptors to this location.
ETH_SEND_1	0x240000	64B	Device driver writes 1 new Send Descriptors to this location. Device driver must pad to 64B.
ETH_SEND_2	0x280000	64B	Device driver writes 2 new Send Descriptors to this location. Device driver must pad to 64B.
ETH_SEND_3	0x2C0000	64B	Device driver writes 3 new Send Descriptors to this

			location. Device driver must pad to 64B.
ETH_RECV_SMALL	0x300000	64B	Device driver writes new Small Receive Buffer Descriptors to this location. Each write must contain 64B of new Descriptors.
ETH_RECV_BIG	0x340000	64B	Device driver writes new Big Receive Buffer Descriptors to this location. Each write must contain 64B of new Descriptors.
ETH_IRQ_DEASSERT	0xF40000	4B	Device driver writes any value to this location to deassert the interrupt.
ETH_CMD	0xF80000	64B	Device driver writes a command to this location to issue various commands to the NIC. See Section 2. Device driver must pad the command to 64B if the command occupies less than 64B.
BOOT_RQST	0xFC0000	64B	Device driver writes a request to this location to perform various operations on the NIC. See Section 3. Unlike ETH_CMD, each request type uses its own offset. Device driver must pad the request to 64B if it occupies less than 64B.

Table 1: Layout of 16MB PCI Memory Space.

2. ETH_CMD Commands

ETH_CMD commands are used primarily during the initialization phase of the device driver. Each command has the format shown in Figure 1.

	0	4
Byte 0	Command	Data0
Byte 8	Data1	Data2
Byte 16	Address MSW	Address LSW

Field Name	Description
Command	Command specifies the type of command.
Data0-2	Data0=2 are used to specify the parameters of a command.
Address MSW	Most significant word of the address of Response Data.
Address LSW	Least significant word of the address of Response Data.

Figure 1: Format of ETH_CMD command.

The device driver first constructs a command in memory and then writes it to offset 0xF80000 of the NIC memory through PIO. The device driver must also reserve 8B for Response Data and Error Code as shown in Figure 2.

	0	4
Byte 0	Response Data	& Error Code

Figure 2: Format of Response Data and Error Code returned by ETH_CMD command.

When the NIC finishes processing the command, it transfers Response Data, if any, to the address specified by Address MSW and Address LSW, and Error Code, if any, to the 4B region immediately following Response Data. While the NIC processes the command, the device driver must wait for the response from the NIC by polling Error Code. Error Code should be initialized to 0xFFFFFFFF. Table 2 describes Error Code.

Name	Error Code	Description
MXGEFW_CMD_OK	0	No error.
MXGEFW_CMD_UNKNOWN	1	Unknown command.
MXGEFW_CMD_ERROR_RANGE	2	Out of range.
MXGEFW_CMD_ERROR_BUSY	3	NIC is busy.

MXGEFW_CMD_ERROR_EMPTY	4	Unused by the Ethernet NIC.
MXGEFW_CMD_ERROR_CLOSED	5	Unused by the Ethernet NIC.
MXGEFW_CMD_ERROR_HASH_ERROR	6	Multicast hash table error.
MXGEFW_CMD_ERROR_BAD_PORT	7	Unused by the Ethernet NIC.
MXGEFW_CMD_ERROR_RESOURCES	8	Unused by the Ethernet NIC.
MXGEFW_CMD_ERROR_MULTICAST	9	Unused by the Ethernet NIC.
MXGEFW_CMD_ERROR_UNALIGNED	10	Unaligned PCI Express packets are detected.

Table 2: Error Code returned by ETH_CMD commands.

Table 3 describes the available ETH_CMD commands.

Command Name	Value	Description
MXGEFW_CMD_NONE	0	No action is taken. Error Code <= MXGEFW_CMD_OK.
MXGEFW_CMD_RESET	1	NIC resets its state. Ethernet link is disabled as a result. If successful, Error Code <= MXGEFW_CMD_OK. If NIC cannot reset its state, Error Code <= MXGEFW_CMD_ERROR_BUSY.
MXGEFW_GET_MCP_VERSION	2	NIC returns version number. Response Data <= Version. Error Code <= MXGEFW_CMD_OK.
MXGEFW_CMD_SET_INTRQ_DMA	3	NIC sets DMA address of Interrupt Queue. Data0: LSW of the address. Data1: MSW of the address. Error Code <= MXGEFW_CMD_OK.
MXGEFW_CMD_SET_BIG_BUFFER_SIZE	4	NIC sets the size of Big Receive Buffer. Data0: Size in bytes. The size must be a power of 2. If successful, Error Code <= MXGEFW_CMD_OK. If NIC detects the size is invalid,

		Error Code <= MXGEFW_ERROR_RANGE.
MXGEFW_CMD_SET_SMALL_BUFFER_SIZE	5	NIC sets the size of Small Receive Buffer. Data0: Size in bytes. Error Code <= MXGEFW_CMD_OK.
MXGEFW_CMD_GET_SEND_OFFSET	6	NIC returns the offset of Send Ring. Response Data <= Offset. Error Code <= MXGEFW_CMD_OK.
MXGEFW_CMD_GET_SMALL_RX_OFFSET	7	NIC returns the offset of Small Receive Buffer Ring. Response Data <= Offset. Error Code <= MXGEFW_CMD_OK.
MXGEFW_CMD_GET_BIG_RX_OFFSET	8	NIC returns the offset of Big Receive Buffer Ring. Response Data <= Offset. Error Code <= MXGEFW_CMD_OK.
MXGEFW_CMD_GET_IRQ_ACK_OFFSET	9	NIC returns the offset of Interrupt Acknowledge. Response Data <= Offset. Error Code <= MXGEFW_CMD_OK.
MXGEFW_CMD_GET_IRQ_DEASSERT_OFFSET	10	NIC returns the offset of Interrupt Deassert. Response Data <= Offset. Error Code <= MXGEFW_CMD_OK.
MXGEFW_CMD_GET_SEND_RING_SIZE	11	NIC returns the size of Send Ring. Response Data <= Size in bytes. Error Code <= MXGEFW_CMD_OK.
MXGEFW_CMD_GET_RX_RING_SIZE	12	NIC returns the size of Big Receive Buffer Ring and Small Receive Buffer Ring. Both Rings have the same size. Response Data <= Size in bytes. Error Code <= MXGEFW_CMD_OK.
MXGEFW_CMD_SET_INTRQ_SIZE	13	NIC sets the size of Interrupt Queue. Data0: Size in bytes. Error Code <= MXGEFW_CMD_OK.
MXGEFW_CMD_ETHERNET_UP	14	NIC attempts to bring up Ethernet link

		and be ready to transfer packets. If successful, Error Code <= MXGEFW_CMD_OK. If NIC encounters an error, Error Code <= MXGEFW_ERROR_RANGE.
MXGEFW_CMD_ETHERNET_DOWN	15	NIC disables Ethernet link and resets its state. Error Code <= MXGEFW_CMD_OK. After Error Code is updated, and when the NIC is ready to reset its state, it sets Link Down field in Interrupt Data to 1 and interrupts the device driver. The device driver must acknowledge this interrupt. Only then the NIC will proceed to reset its state. In order to bring up the NIC to a usable state again, the device driver should perform the initialization steps starting at step 6 in Section 4.
MXGEFW_CMD_SET_MTU	16	NIC sets MTU. The NIC's MTU is the maximum size of an Ethernet packet including the Ethernet header and payload minus the 4B CRC. Data0: MTU. If successful, Error Code <= MXGEFW_CMD_OK. If the NIC cannot change MTU, Error Code <= MXGEFW_ERROR_RANGE. The device driver must not change MTU after the initialization.
MXGEFW_CMD_GET_INTR_COAL_DELAY_OFFSET	17	NIC returns the offset of Interrupt Coalescing Delay. Response Data <= Offset. Error Code <= MXGEFW_CMD_OK.
MXGEFW_CMD_SET_STATS_INTERVAL	18	NIC sets Statistics Update Interval. This command currently has no effect. Error Code <= MXGEFW_CMD_OK.

MXGEFW_CMD_SET_STATS_DMA_OBSOLETE	19	Obsolete.
MXGEFW_ENABLE_PROMISC	20	NIC enables Promiscuous Mode. Code <= MXGEFW_CMD_OK.
MXGEFW_DISABLE_PROMISC	21	NIC disables Promiscuous Mode. Error Code <= MXGEFW_CMD_OK.
MXGEFW_SET_MAC_ADDRESS	22	NIC sets the MAC address. Data0: Byte 0: 1st octet of MAC address. Byte 1: 2nd octet of MAC address. Byte 2: 3rd octet of MAC address. Byte 3: 4th octet of MAC address. Data1: Byte 2: 5th octet of MAC address. Byte 3: 6th octet of MAC address. Error Code <= MXGEFW_CMD_OK.
MXGEFW_ENABLE_FLOW_CONTROL	23	NIC enables Ethernet Flow Control. NIC also enables Block on Receive Buffer. Error Code <= MXGEFW_CMD_OK.
MXGEFW_DISABLE_FLOW_CONTROL	24	NIC disables Ethernet Flow Control. NIC also disables Block on Receive Buffer. Error Code <= MXGEFW_CMD_OK.
MXGEFW_DMA_TEST	25	NIC performs a DMA test. Data0: LSW of DMA address. Data1: MSW of DMA address. Data2: Bytes 0 1: Read DMA length in bytes. Bytes 2 3: Write DMA length in bytes. Response Data <= Bytes 0 1: Number of DMA operations. Bytes 2 3: Number of 1/2us ticks. If the test completes, Error Code <= MXGEFW_CMD_OK. If NIC cannot perform the test, Error Code <= MXGEFW_CMD_ERROR_RANGE.

MXGEFW_ENABLE_ALLMULTI	26	NIC enables multicast filtering. Error Code <= MXGEFW_CMD_OK.
MXGEFW_DISABLE_ALLMULTI	27	NIC disables multicast filtering. Error Code <= MXGEFW_CMD_OK.
MXGEFW_JOIN_MULTICAST_GROUP	28	NIC inserts an address to the multicast filtering table. Data0: Byte 0: 1st octet of MAC address. Byte 1: 2nd octet of MAC address. Byte 2: 3rd octet of MAC address. Byte 3: 4th octet of MAC address. Data1: Byte 0: 5th octet of MAC address. Byte 1: 6th octet of MAC address. If successful, Error Code <= MXGEFW_CMD_OK. If NIC cannot insert the address, Error Code <= MXGEFW_CMD_ERROR_HASH_ERROR.
MXGEFW_LEAVE_MULTICAST_GROUP	29	NIC removes an address from the multicast filtering table. Data0: Byte 0: 1st octet of MAC address. Byte 1: 2nd octet of MAC address. Byte 2: 3rd octet of MAC address. Byte 3: 4th octet of MAC address. Data1: Byte 0: 5th octet of MAC address. Byte 1: 6th octet of MAC address. If successful, Error Code <= MXGEFW_CMD_OK. If NIC cannot insert the address, Error Code <= MXGEFW_CMD_ERROR_HASH_ERROR.
MXGEFW_LEAVE_ALL_MULTICAST_GROUPS	30	NIC removes all entries from the

		multicast filtering table. If successful, Error Code <= MXGEFW_CMD_OK. If NIC cannot remove all entries, Error Code <= MXGEFW_CMD_ERROR_HASH_ERROR.
MXGEFW_CMD_SET_STATS_DMA_V2	31	NIC sets the DMA address and size of Interrupt Data. Data0: LSW of the address. Data1: MSW of the address. Data2: Size in bytes. Error Code <= MXGEFW_CMD_OK.
MXGEFW_CMD_UNALIGNED_TEST	32	Same as MXGEFW_DMA_TEST except that NIC stops the test as soon as it detects unaligned PCI Express packets. If such packets are detected, Error Code <= MXGEFW_CMD_ERROR_UNALIGNED
MXGEFW_CMD_UNALIGNED_STATUS	33	NIC reports whether it has detected unaligned PCI Express packets. If no such packets are detected, Error Code <= MXGEFW_CMD_OK. If such packets are detected, Error Code <= MXGEFW_CMD_ERROR_UNALIGNED

Table 3: Available ETH_CMD commands.

3. BOOT_RQST Commands

BOOT_RQST commands are used during an early stage of the device driver initialization. Each command has the format shown in Figure 3.

	0	4
Byte 0	Data0	Data1
Byte 8	Data2	Data3
…		
Byte 56	Data14	Data15

Figure 3: Format of BOOT_RQST request.

The device driver first constructs a 64B command in memory and then writes it to the offset of the NIC memory that is assigned to the command through PIO. Each type of command may interpret Data fields differently. Like ETH_CMD commands, the NIC usually notifies the device driver of the completion by transferring certain values to main memory. Refer to Section 2 on how the driver allocates space for completion notices. Table 4 describes the available BOOT_RQST commands.

Command Name	Offset (B)	Description
BOOT_HANDOFF (MXGEFW_BOOT_HANDOFF)	0xFC0000	NIC attempts to perform handoff of firmware. See Section 4.1. Data0: MSW of DMA address of confirmation. Data1: LSW of DMA address of confirmation. Data2: Confirmation data. Data3: NIC address of new firmware. Data4: Size of new firmware in bytes. Data5: NIC address of relocation address. Data6: NIC address of the start instruction. NIC copies the new firmware specified by Data3 and Data4 to a new location specified by Data5. NIC then starts executing the instruction specified by Data6. Upon a successful handoff, the NIC transfers the confirmation data specified by Data2 to the location specified by Data0 and Data1.
BOOT_DUMP	0xFC0040	NIC transfers an arbitrary region of the NIC

		memory to main memory buffer specified by the device driver. Data0: MSW of DMA address of the buffer. Data1: LSW of DMA address of the buffer. Data2: NIC address. Data3: Size in bytes. Data4: MSW of DMA address of confirmation. Data5: LSW of DMA address of confirmation. Data6: Confirmation data. Data0 and Data1 specify the main memory buffer. Data2 and Data3 specify the NIC memory region to be copied. Upon completion of the copy, NIC transfers the confirmation data specified by Data6 to the location specified by Data4 and Data5.
BOOT_END	0xFC0080	NIC returns the end address of the firmware. Data0: MSW of DMA address of confirmation. Data1: LSW of DMA address of confirmation. The NIC transfers the end address to the location specified by Data0 and Data1.
BOOT_JUMP	0xFC00C0	NIC executes a function and returns its return value. Data0: MSW of DMA address of confirmation. Data1: LSW of DMA address of confirmation. Data2: Address of the function to call. Upon return from the function, the NIC transfers the return value to the location specified by Data0 and Data1.
BOOT_EEPROM_CTRL	0xFC0100	NIC accesses JTAG. Data0: Control data.
BOOT_EEPROM_ADDR_DATA	0xFC0140	NIC accesses JTAG. Data0: JTAG address. Data1: JTAG data. Data2: MSW of DMA address of confirmation. Data3: LSW of DMA address of confirmation. Data4: Flag.

		If Flag specified by Data4 is non-zero, the NIC transfers the result of the JTAG access to the location specified by Data2 and Data3.
BOOT_JTAG_RELEASE	0xFC0180	No action is taken.
BOOT_DUMMY_RDMA (MXGEFW_BOOT_DUMMY_RDMA)	0xFC01C0	NIC performs a read DMA. Data0: MSW of DMA address of confirmation. Data1: LSW of DMA address of confirmation. Data2: Confirmation data. Data3: MSW of read DMA address. Data4: LSW of read DMA address. Data5: Read DMA enable. If Data5 specifies non-zero value, then NIC periodically, performs a read DMA of 1B from the location specified by Data3 and Data4. The NIC stops periodic read DMAs if the device driver issues another BOOT_DUMMY_RDMA with Data5 set to zero. NIC transfers the confirmation data specified by Data2 to the location specified by Data0 and Data1.

Table 4: Available BOOT_RQST commands.

4. Initializing the NIC

Upon a system reset, the NIC is running the firmware extracted from the on-board EEPROM. At minimum, this firmware is able to handle PCI Express traffic and implements BOOT_RQST commands (see Section 3).

The device driver initializes the NIC through a series of commands. The purpose of the initialization is to bring the NIC to the state in which it can start sending and receiving Ethernet packets. In general, the device driver should follow the sequence of steps shown below. The exact mechanism used to initialize the device driver itself is specific to the operating system and is not discussed here.

1. Identify the PCI device. Myricom's PCI Vendor ID is 0x14C1. The current 10 Gigabit Ethernet NICs use PCI Device ID 0x0008.
2. Modify PCI configuration if necessary. The device driver should enable the bus master. It may enable or disable other features such as MSI.
3. Map the device's PCI memory space into memory region that is directly accessible by the driver.
4. Extract any driver-specific bookkeeping information from mcp_gen_header (Section 8.2) and STRING_SPECS (Section 8.3). For instance, the device driver reads the NIC's default MAC address from STRING_SPECS.
5. Download a new version of the firmware if necessary. See Section 4.1. This new firmware must be a fully-functional Ethernet firmware.
6. Issue MXGEFW_CMD_RESET to initialize the NIC state.
7. Allocate Interrupt Queue (Section 6.6) and inform the NIC of its location and size through MXGEFW_CMD_SET_INTRQ_SIZE and MXGEFW_CMD_SET_INTRQ_DMA.
8. Read the Interrupt Acknowledge offset through MXGEFW_CMD_GET_IRQ_ACK_OFFSET and save it for future use.
9. Read the interrupt deassert offset through MXGEFW_CMD_GET_IRQ_DEASSERT_OFFSET and save it for future use. This offset is used only if the device driver uses the legacy INTx interrupt mode.
10. Read the offset of the interrupt coalescing interval period through MXGEFW_CMD_GET_INTR_COAL_DELAY_OFFSET. Set an appropriate period in microseconds by writing the value to the offset obtained through the command.
11. If desired, perform DMA tests using MXGEFW_DMA_TEST.
12. Set the MAC address through MXGEFW_SET_MAC_ADDRESS. The device driver must set the MAC address through this command during the initialization.

13. Disable the promiscuous mode through MXGEFW_DISABLE_PROMISC.
14. If desired, enable Ethernet flow control through MXGEFW_ENABLE_FLOW_CONTROL.
15. Read the offset of Send Ring (Section 5.6) through MXGEFW_CMD_GET_SEND_OFFSET save it for future use.
16. Read the offset of Small Receive Buffer Ring (Section 6.4) through MXGEFW_CMD_GET_SMALL_RX_OFFSET and save it for future use.
17. Read the offset of Big Receive Buffer Ring (Section 6.4) through MXGEFW_CMD_GET_BIG_RX_OFFSET and save it for future use.
18. Set MTU through MXGEFW_CMD_SET_MTU.
19. Allocate Receive Buffers and update Receive Buffer Rings in the NIC memory.
20. Set the size of Small Receive Buffer through MXGEFW_CMD_SET_SMALL_BUFFER_SIZE.
21. Set the size of Big Receive Buffer through MXGEFW_CMD_SET_BIG_BUFFER_SIZE.
22. Inform the NIC of the location and size of Interrupt Data in main memory through MXGEFW_CMD_SET_STATS_DMA_V2.
23. Issue MXGEFW_CMD_ETHERNET_UP. Once this command completes successfully, the NIC is ready to start sending and receiving packets. At this point, the device driver must be able to process interrupts.

4.1. Firmware Download

The NIC implements a mechanism that enables the device driver to download a new version of the firmware and to start the new firmware. In order to download the firmware, the device driver must follow the steps below. The steps are only a convention and may change at any time. The exact binary format of the firmware is outside the scope of this document. Contact Myricom for information.

1. Copy the binary firmware image to the NIC memory starting at 0x100000 (1MB) through PIO.
2. Issue MXGEFW_BOOT_HANDOFF along with the location of the new firmware (0x100000), size of the new firmware, the new location (0x0), and the expected return value. Upon receiving the command, the NIC copies the image of the new firmware to the new location and starts executing it. The new firmware then transfers the return value specified by the driver to the main memory location, also specified by the driver.
3. Check for the expected return value. If the expected value is found, then the download process is complete, and the NIC is running the new firmware. If the NIC does not transfer the expected value within a second, then the device driver should consider the NIC

is in an unusable state. The one second timeout is a conservative value. Handoff process typically takes much shorter.

Note that the device driver cannot download arbitrary firmware.

4.2. Resetting the NIC

In order to gracefully reset the NIC, the device driver should use MXGEFW_CMD_ETHERNET_DOWN. When the device driver issues this command, the NIC returns a success code MXGEFW_CMD_OK and then attempts to complete any pending tasks. Once the NIC is ready to reset its state, it sets Link Down in Interrupt Data and generates an interrupt. The device driver must process this interrupt as a normal interrupt. The NIC only resets its state after the interrupt has been processed by the driver.

In order to bring up the NIC to a usable state again, the device driver should perform the initialization steps described in Section 4 starting at step 6.

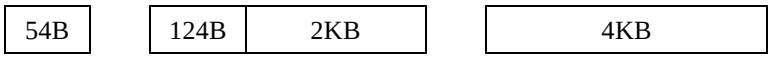
5. Sending Packets

The host creates packets to be sent in main memory. The device driver creates descriptors that contain the locations and sizes of the packets along with any other operations that the NIC must perform to the packet. These descriptors are then transferred to the NIC memory. The NIC examines the descriptors, fetches the packets, and then transmits them. Upon packet transmission, the NIC notifies the device driver of the number of packets that have been sent.

The NIC supports checksum offload, TCP segmentation offload (TSO), and jumbo packets.

5.1. Packets and Segments

A packet consists of a series of physically contiguous main memory buffers. In order to send a packet, the device driver must divide these buffers into a series of segments and notify the NIC of each segment. A segment is a physically contiguous main memory region that is contained within a 4KB page. The diagram below illustrates the relationship between a packet and the segments that belong to the packet. Assume that the second buffer crosses the 4KB page boundary.



Physically contiguous buffers that belong to a packet.

Segments that belong to a packet.

There is one exception to the definition of segment above. Some versions of the NIC firmware require that the segment size be at most 2KB, not 4KB. For these firmware versions, the device driver must ensure that a segment is a contiguous memory within a 2KB region and does not cross 2KB boundaries.

5.2. TCP Segmentation Offload

The NIC supports TCP segmentation offload (TSO). Using TSO, the operating system creates a TCP packet whose payload size is much larger than maximum segment size (MSS). Currently, the packet can be as large as 64KB. The NIC breaks the large packet into a series of smaller TCP packets (TSO-segmented packets) whose payloads are at most MSS bytes.

TCP headers of the TSO-segmented packets carry the same control bits as the original TSO packet. The exception is PUSH and FIN flags. PUSH and FIN are removed from the TSO-segmented packets, except the last packet.

Currently, the NIC does not support TSO when IPv6 is used or when the Ethernet header includes a VLAN tag.

5.3. Checksum Offload

For regular (non-TSO) packets, the NIC supports TCP and UDP checksum offload. The NIC can compute TCP and UDP checksums and correctly place them in the checksum field of TCP or UDP header. This feature is optional, and the device driver uses a flag to instruct the NIC to perform checksum calculations, on a packet-by-packet basis. The NIC assumes that the operating system computes IP checksum and pseudo header checksum (for TCP or UDP packets). The IP header must have the correct IP header checksum. The checksum field in TCP or UDP header must contain the checksum of the pseudo header. The NIC correctly computes TCP/UDP checksums whether IPv4 or IPv6 is used, because the operating system computes IP and TCP/UDP pseudo header checksums.

TSO packets must also have correct IP and pseudo header checksums in their respective fields. The NIC uses these checksums as a basis to compute final checksums for individual TSO-segmented packets it generates.

5.4. Send Segment Descriptor

Send Segment Descriptor is a 16B data structure that contains information about a segment of a packet to be sent. The NIC uses the information stored in the Descriptor in order to initiate DMA operations and to transmit the packet. Send Segment Descriptor has the fields shown in Table 5.

Field Name	Offset (B)	Size (B)	Description
MSW Address	0	4	MSW of the start DMA address of the segment.
LSW Address	4	4	LSW of the start DMA address of the segment.
Pseudo Header Offset	8	2	For non-TSO packets, this field contains the offset, from the start of the packet, to the start of Checksum field of the TCP/UDP packet. The NIC uses this field only when checksum offload is requested by the driver (i.e. MXGEFW_FLAGS_CKSUM is set in Flags). Currently, the offset cannot be greater than 134. Otherwise, the driver must compute checksums. Note that for a valid TCP/UDP packet with an IPv4 header,

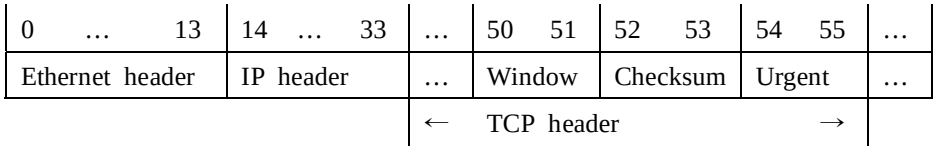
			the offset is guaranteed to be less than 134. For TSO packets, this field contains the maximum segment size (MSS).
Length	10	2	Length of the segment in bytes.
Pad	12	1	This field must be set to 0.
DMA Count	13	1	For regular (non-TSO) packets, this field contains the number of segments (read DMA operations) that belong to the packet. The NIC uses this field to determine the end of the packet. For TSO packets, if the segment starts a new TSO-segmented packet, then this field contains the number of segments (read DMA operations) that belongs to the TSO-segmented packet. The NIC uses this field to determine the end of the TSO-segmented packet.
Checksum Offset	14	1	Offset, from the start of the segment, to the start of the payload of the IP packet. So, this offset corresponds to the start of the TCP/UDP packet within the IP packet. The NIC uses this field only when checksum offload is requested by the driver (i.e. MXGEFW_FLAGS_CKSUM is set in Flags). The offset may be larger than Length, indicating that the IP payload starts in a subsequent segment. If the offset is negative, indicating that the IP payload starts in a previous segment, then the offset must be set to zero. The offset cannot be greater than 255 as the field is limited to 1B. Otherwise, the driver must compute checksums.
Flags	15	1	Flags. See Table 6.

Table 5: Format of Send Segment Descriptor.

Using C code, Send Descriptor may be declared as follows.

```
struct mcp_kreq_ether_send {
  uint32_t addr_high;
  uint32_t addr_low;
  uint16_t pseudo_hdr_offset;
  uint16_t length;
  uint8_t  pad;
  uint8_t  rdma_count;
  uint8_t  cksum_offset;
  uint8_t  flags;
};
```

The following shows an example of an Ethernet packet.



In this case, Checksum Offset of Send Segment Descriptor should be set to 34 since the TCP packet starts at byte 34. Pseudo Header Offset of Send Segment Descriptor should be set to 52 since Checksum field of the TCP header starts at byte 52. As mentioned previously, Checksum field must already contain the TCP pseudo header checksum.

Flag Name	Value	Description
MXGEFW_FLAGS_SMALL	0x1	This flag is set if the packet length is at most MXGEFW_SEND_SMALL_SIZE (1520B).
MXGEFW_FLAGS_TSO_HDR	0x1	This flag is set on the first Send Segment Descriptor for a TSO packet. The first segment contains only the headers (Ethernet, IP, and TCP).
MXGEFW_FLAGS_FIRST	0x2	This flag is set if the segment starts a new packet.
MXGEFW_FLAGS_ALIGN_ODD	0x4	For non-TSO packets, this flag is set if the total length of the previous segments is odd. This flag is required only if checksum offload is used. For TSO packets, this flag is set if the total length of the previous segments that belong to the current TSO-segmented packet is odd.

MXGEFW_FLAGS_CKSUM	0x8	For non-TSO packets, this flag is set if the NIC must compute TCP/UDP checksums. For TSO packets, this flags is not used.
MXGEFW_FLAGS_TSO_LAST	0x8	For TSO packets, this flag is set if the segment is the last segment. For non-TSO packets, this flag is not used.
MXGEFW_FLAGS_NO_TSO	0x10	This flag is set if the packet is not a TSO packet.
MXGEFW_FLAGS_TSO_CHOP	0x10	For TSO packets, this flag is set if the segment spans more than one TSO-segmented packets. For non-TSO packets, this flag is not used.
MXGEFW_FLAGS_TSO_PLD	0x20	For TSO packets, this flag is set on all segments except the first segment that only contains the headers. For non-TSO packets, this flags is not used.

Table 6: Flags used in Send Segment Descriptor.

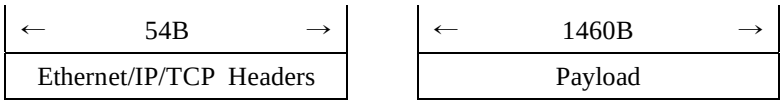
5.5 Rules for Generating Send Segment Descriptor

For a regular, non-TSO packet, the device driver must use the rules shown below to generate Send Segment Descriptors. The device driver first breaks the packet into a list of segments. As previously mentioned, a segment is a physically contiguous main memory region that is contained within a 4KB page. The device driver then creates one Send Segment Descriptor for each segment as follows.

1. Set MXGEFW_FLAGS_NO_TSO in every Descriptor.
2. Set MXGEFW_FLAGS_FIRST in the first Descriptor.
3. If the entire packet length is at most MXGEFW_SEND_SMALL_SIZE (1520B), then set MXGEFW_FLAGS_SMALL in every Descriptor.
4. If TCP/UDP checksum offload is used, then set MXGEFW_FLAGS_CKSUM in every Descriptor.
5. If TCP/UDP checksum offload is used, then set MXGEFW_FLAGS_ALIGN_ODD in each Descriptor if the cumulative sum of Length in all the previous Descriptors is odd. So, MXGEFW_FLAGS_ALIGN_ODD is never set in the first Descriptor.
6. Set MSW Address and LSW Address in each Descriptor to be the DMA address of the start of the segment.
7. Set Length in each Descriptor to be the length of the segment in bytes.
8. Set DMA Count in the first Descriptor to be the total number of the segments.
9. If TCP/UDP checksum offload is used, then set Pseudo Header Offset in the first Descriptor to be the offset of the checksum field in TCP or UDP header.

10. If TCP/UDP checksum offload is used, then set Checksum Offset in each Descriptor as follows.
    A. If the start of the IP payload (the start of TCP or UDP header) appears in a previous segment, then set Checksum Offset to be 0.
    B. If the start of the IP payload appears in the current segment, then set Checksum Offset to be the offset from the start of the current segment to the start of the IP payload. Thus, the value is equal to the number of bytes in the current segment that appear before the start of the IP payload.
    C. If the start of the IP payload appears in a subsequent segment, then set Checksum Offset to be the offset from the start of the current segment to the start of the IP payload. Thus, the value is equal to the number of bytes that appear between the start of the current segment and the start of the IP payload. Note that this value is always greater than Length in the Descriptor.

The diagram below shows an example of regular packet and the Descriptors generated for the packet. The packet consists of two segments shown as the boxes. Assume that Ethernet header is 14B, IP header is 20B, and TCP header is 20B, and that TCP checksum offload is used.



1st Send Segment Descriptor

Pseudo Header Offset: 52

Checksum Offset: 34

DMA Count: 2

Length: 54

Flags: MXGEFW_FLAGS_SMALL
MXGEFW_FLAGS_CKSUM
MXGEFW_FLAGS_NO_TSO
MXGEFW_FLAGS_FIRST

2nd Send Segment Descriptor

Pseudo Header Offset: 0 (Not used by the NIC)

Checksum Offset: 0

DMA Count: 1 (Not used by the NIC)

Length: 1460

Flags:	MXGEFW_FLAGS_SMALL
	MXGEFW_FLAGS_CKSUM
	MXGEFW_FLAGS_NO_TSO

For a TSO packet, the device driver must use the rules shown below to generate Send Segment Descriptors. The device driver first breaks the packet into the header part and the payload part. The header part consists of the Ethernet header, IP header, and TCP header. The payload part consists of the entire TCP payload. The device driver then breaks each part into a list of segments and creates one Send Segment Descriptor for each segment. The two segment lists combined (the header segments followed by the payload segments) represent the TSO packet.

For the header segments, the device driver must use the following rules.

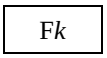
1. Set MXGEFW_FLAGS_TSO_HDR in each Descriptor.
2. Set MXGEFW_FLAGS_FIRST in the first Descriptor.
3. Set MSW Address and LSW Address in each Descriptor to be the DMA address of the start of the segment.
4. Set Pseudo Header Offset in each Descriptor to be the maximum segment size (MSS).
5. Set Length in each Descriptor to be the length of the segment in bytes.
6. Set DMA Count in the first Descriptor to be the total number of the segments (only the header segments).

For the payload segments, the device driver must use the following rules.

1. Set MXGEFW_FLAGS_TSO_PLD in every Descriptor.
2. Set MXGEFW_FLAGS_SMALL in every Descriptor if the maximum segment size (MSS) is at most MXGEFW_SEND_SMALL_SIZE (1520B).
3. Set MXGEFW_FLAGS_TSO_LAST in the last Descriptor.
4. Set MSW Address and LSW Address in each Descriptor to be the DMA address of the start of the segment.
5. Set Pseudo Header Offset in each Descriptor to be the maximum segment size (MSS).
6. Set Length in each Descriptor to be the length of the segment in bytes.
7. Set the rest of the fields using the following rules. The device driver first breaks the payload segments into TSO-segmented payloads. As mentioned previously, each TSO-segmented payload has MSS bytes, except the last payload, which may have less than MSS bytes.

   A. The segment contains a part of one TSO-segmented payload. The diagram below

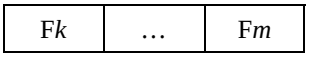
illustrates this case, where the box represents the segment and F*k* represents a part of the TSO-segmented payload P*k*.



If F*k* starts P*k*, then set MXGEFW_FLAGS_FIRST in the Descriptor and also set DMA Count of the Descriptor to be the number of segments that P*k* spans.

If the number of bytes that belong to P*k* and precede F*k* is odd, then set MXGEFW_FLAGS_ALIGN_ODD in the Descriptor.

B. The segment contains parts of two or more TSO-segmented payloads. The diagram below illustrates this case, where the box represents the segment and F*k* and F*m* represent parts of two TSO-segmented payloads P*k* and P*m*, respectively.



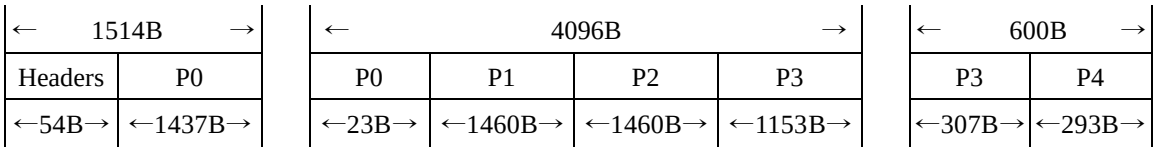
Set MXGEFW_FLAGS_TSO_CHOP in the Descriptor.

Set DMA Count of the Descriptor to be the number of segments that P*m* spans.

If F*k* starts P*k*, then set MXGEFW_FLAGS_FIRST in the Descriptor.

If the number of bytes that belong to P*k* and precede F*k* is odd, then set MXGEFW_FLAGS_ALIGN_ODD.

The diagram below shows an example of TSO packet and the Descriptors generated for the packet. The TSO packet consists of three segments, shown as the boxes. P*i* represents a TSO-segmented payload. Assume the MSS is 1460B. In the Descriptors, only DMA Count, Length, and Flags are shown, as it is trivial to compute the values for the other fields using the rules above.



1st Send Segment Descriptor (Headers)

DMA Count: 1

Length: 54

Flags: MXGEFW_FLAGS_TSO_HDR
MXGEFW_FLAGS_FIRST

2nd Send Segment Descriptor (P0)

DMA Count: 2

Length: 1437

Flags: MXGEFW_FLAGS_SMALL

MXGEFW_FLAGS_FIRST

MXGEFW_FLAGS_TSO_PLD

3rd Send Segment Descriptor (P0, …, P3)

DMA Count: 2

Length: 4096

Flags: MXGEFW_FLAGS_SMALL

MXGEFW_FLAGS_TSO_PLD

MXGEFW_FLAGS_TSO_CHOP

MXGEFW_FLAGS_ALIGN_ODD

4th Send Segment Descriptor (P3, P4)

DMA Count: 1

Length: 600

Flags: MXGEFW_FLAGS_SMALL

MXGEFW_FLAGS_TSO_PLD

MXGEFW_FLAGS_TSO_CHOP

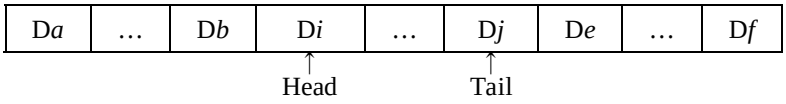
MXGEFW_FLAGS_ALIGN_ODD

MXGEFW_FLAGS_TSO_LAST

5.6. Send Ring

Send Ring is a ring of Send Segment Descriptors. The device driver produces new Descriptors in order to alert the NIC that there are new packets to be sent. The NIC uses the information in the Descriptors to fetch and transmit packets. The ring resides in the NIC memory. The device driver uses MXGEFW_CMD_GET_SEND_OFFSET and MXGEFW_CMD_GET_SEND_RING_SIZE to determine the location and the size of the ring in the NIC memory during the initialization.

In the NIC memory, Send Ring is one contiguous array of Send Segment Descriptors as shown in Figure 4.



D*a*... D*b*	Invalid Send Segment Descriptors. Flags is 0.
D*i* ... D*j*	Valid Send Segment Descriptors that contain information about segments (packets) to be sent. Note that Flags cannot be 0.
D*e* ... D*f*	Invalid Send Segment Descriptors. Flags is 0.

Figure 4: Send Ring.

Both the head and tail pointers are initialized to the first entry of the ring D*a*. The NIC initializes the entire ring with all 0s to indicate that the ring is empty. The NIC keeps track of only the head pointer and infers the tail pointer by examining Flags of Descriptors. The device driver keeps track of both the head and tail pointers.

The NIC polls the Descriptor pointed by its head pointer and advances the head pointer when it processes the Descriptor. The device driver advances its tail pointer when it produces new Send Segment Descriptors for packets to be sent. It creates Send Segment Descriptors for the packets and then copies them to Send Ring in the NIC memory (see below for the ways of copying Descriptors). The device driver advances its head pointer when it processes (e.g. de-allocates) transmitted packets. The NIC notifies the device driver the number of sent packets in Send Done Count of Interrupt Data. Based on this number, the device driver computes the number of Send Segment Descriptors consumed by the NIC and advances its head pointer by the same number.

There are two ways for the device driver to copy Send Segment Descriptors to Send Ring in the NIC memory.

1. The device driver may copy Descriptors directly to their corresponding entries in the NIC memory. To do so, the driver first computes the locations of the corresponding entries of the ring in the NIC memory based on its tail pointer as well as the offset and the size of the ring obtained through MXGEFW_CMD_GET_SEND_OFFSET and MXGEFW_CMD_GET_SEND_RING_SIZE. Then, the driver writes Descriptors to those locations through PIO.

   If this method is used, the first Descriptor must be written last.
2. Alternatively, the NIC may copy the Descriptors to a 64B region starting at the offset

ETH_SEND_1, ETH_SEND_2, ETH_SEND_3, or ETH_SEND_4. The NIC then places the new Descriptors into the correct entries in Send Ring.

A. In order to copy 4 Descriptors, the device driver uses ETH_SEND_4.

B. In order to copy 3 Descriptors, the device driver uses ETH_SEND_3.

C. In order to copy 2 Descriptors, the device driver uses ETH_SEND_2.

D. In order to copy 1 Descriptors, the device driver uses ETH_SEND_1.

To use this method, the device driver must always fill the entire 64B region and pad bytes if necessary. For example, to copy 3 Descriptors, which require 48B, the driver must pad 16B.

This method is intended for a system that allows the device driver to enable write-combining for PIO accesses. In such a system, the device driver should use write-combining to ensure that all 64B are transferred to the NIC in a single PCI Express packet. For instance, the device driver may use various instructions that serve as memory barrier and flush the store buffer of the processor once all 64B are written. Note that even if the system generates multiple PCI Express packets with smaller-than-64B payloads, the NIC correctly combines them into a contiguous 64B of Descriptors. However, it generally leads to performance degradations, and the device driver should use the first method to copy Descriptors.

5.7. Operations

The device driver should follow the steps below to transmit a given packet.

1. If the packet is smaller than 60B, then pad bytes so that the packet is at least 60B. The pad bytes must be zeros. The NIC does not automatically pad small packets to the minimum size.
2. Create Send Segment Descriptors for the packet.
3. Check whether Send Ring has enough free slots to insert the new Descriptors. If there are not enough slots, then terminate the transmit attempt.
4. Copy the Descriptors to Send Ring.
5. Advance the device driver's tail pointer of Send Ring by the number of the Descriptor.
6. Store bookkeeping information about the packet so that the driver can de-allocate it when it is sent.

The NIC always transmits packets in the order specified in Send Ring. When a packet is sent, the NIC increments Send Done Count in Interrupt Data and may generate an interrupt. In order to process sent packets, the device driver should follow the steps below.

1. Determine how many packets are sent based on Send Done Count and any driver-specific

bookkeeping information.

2. Determine how many Send Segment Descriptors have been consumed by the NIC, based on the number of packets sent. The device driver must implement a mechanism to determine how many Descriptors are used for each packet sent.
3. De-allocate the sent packets.
4. Advance the driver's head pointer of Send Ring by the number of Send Segment Descriptors consumed by the NIC.

6. Receiving Packets

The device driver provides the NIC with main memory buffers. When the NIC receives packets from the link, it transfers them into these buffers. Upon processing received packets, the driver must allocate new buffers and notify the NIC. The NIC supports checksum offload for received packets. It also supports jumbo packets.

6.1. Receive Buffer

Receive Buffer is a main memory buffer allocated by the device driver. The NIC transfers received packets into Receive Buffers. There are two types of Receive Buffer.

Small Receive Buffer:

A Small Receive Buffer always contains one complete packet. The size of Small Receive Buffer must be a multiple of 4B and be at most 4KB. Each buffer must reside entirely within a 4KB page. A Small Receive Buffer must also start at a 4B-aligned address. The device driver notifies the NIC of the buffer size through MXGEFW_CMD_SET_SMALL_BUFFER_SIZE. If the length of the received packet is at most the buffer size, then the NIC stores the packet in Small Receive Buffer.

Because the NIC implicitly skips the first 2 bytes and stores the packet starting at byte offset 2, the actual size of Small Receive Buffer must be the size advertised through MXGEFW_CMD_SET_SMALL_BUFFER_SIZE plus 2. For example, if the device driver advertises the size 1514B through MXGEFW_CMD_SET_SMALL_BUFFER_SIZE, then it must allocate at least 1516B for each Small Receive Buffer.

Big Receive Buffer:

The size of Big Receive Buffer must be at least 4KB and must be a power of 2. A Big Receive Buffer must also start at a 4B-aligned address. The device driver notifies the NIC of the buffer size through MXGEFW_CMD_SET_BIG_BUFFER_SIZE. If the received packet is larger than the size of Small Receive Buffer (advertised through MXGEFW_CMD_SET_SMALL_BUFFER_SIZE), then the NIC stores the packet into Big Receive Buffer(s). If the packet requires more than one buffer, then it will span across multiple buffers.

The intended use of Small Receive Buffer is to receive Ethernet packets with the standard MTU. For instance, the device driver would allocate 1520B Small Receive Buffers, 9400B Big Receive Buffers, and advertise 1514B to the NIC through MXGEFW_CMD_SET_SMALL_BUFFER_SIZE. Jumbo packets that are larger than 1514B will then be received into Big Receive Buffers.

The device driver is free to use other sizes for Small and Big Receive Buffers. For instance, if the NIC's MTU is 1514B, then the device driver may use both Small and Big Receive Buffers to receive Ethernet packets with the standard MTU. The driver may allocate 128B Small Receive Buffers to receive very small packets, and allocate 1520B Big Receive Buffers to receive larger packets. In this case, the driver still advertises 4KB through MXGEFW_CMD_SET_BIG_BUFFER_SIZE. However, because the NIC never writes data past the end of packet, 1520B buffers are sufficient.

6.2. Received Packet Layout

A received packet in a Receive Buffer includes a complete Ethernet frame except the CRC. The packet starts at offset 2 (so the first two bytes of Received Buffer are unused) so that the IP header, if present, starts at a 4B-aligned address. Figure 5 shows a received packet in a Receive Buffer.

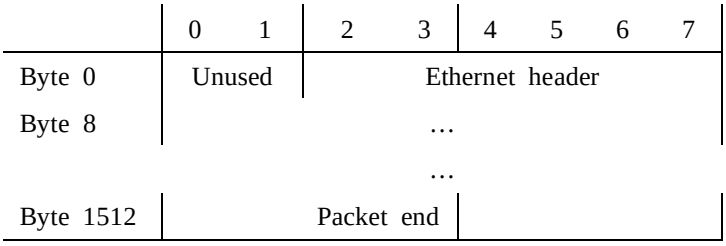
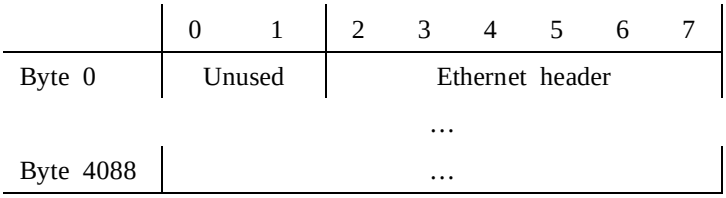
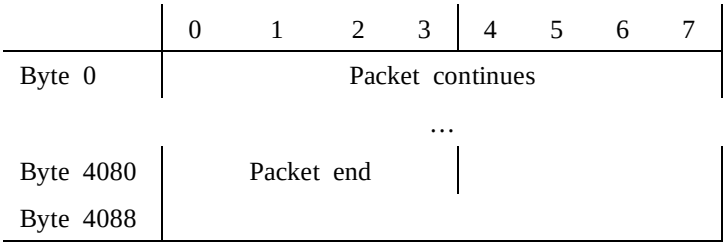


Figure 5: Received packet of size 1514B in a 1520B Receive Buffer.

A jumbo packet may span multiple Big Receive Buffers as illustrated in Figure 6.



First Big Receive Buffer (4KB).



Second Big Receive Buffer (4KB).

Figure 6: Received jumbo packet of size 8178B spanning two 4KB Big Receive Buffers.

6.3. Receive Buffer Descriptor

Receive Buffer Descriptor is a 8B data structure that indicates the start address of a Receive Buffer. This Descriptor is used for both Small Receive Buffer and Big Receive Buffer. The Descriptor format is shown in Table 7.

Field Name	Offset (B)	Size (B)	Description
MSW Address	0	4	MSW of the start address of Receive Buffer.
LSW Address	4	4	LSW of the start address of Receive Buffer.

Table 7: Receive Buffer Descriptor.

If the MSW of the address is zero (for instance, because the machine uses 32-bit addresses), then it must be filled with zeros.

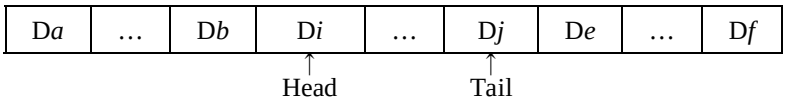
Using C code, Receive Buffer Descriptor may be declared as follows.

```
struct mcp_kreq_ether_recv {
  uint32_t addr_high;
  uint32_t addr_low;
};
```

6.4. Receive Buffer Ring

Receive Buffer Ring is a ring of Receive Buffer Descriptors, describing the locations of free main memory buffers that the NIC can use to store received packets. The NIC consumes Receive Buffers specified in Receive Buffer Rings when it transfers received packets, and the device driver produces (replenishes) new Receive Buffers. The NIC has two Receive Buffer Rings in the NIC memory: Small Receive Buffer Ring that stores the locations of Small Receive Buffers and Big Receive Buffer Ring that stores the locations of Big Receive Buffers. Both rings reside in the NIC memory. During the initialization, the device driver determines the location and size of these rings through MXGEFW_CMD_GET_SMALL_RX_OFFSET, MXGEFW_CMD_GET_BIG_RX_OFFSET, and MXGEFW_CMD_GET_RX_RING_SIZE.

In the NIC memory, a Receive Buffer Ring is one contiguous array of Receive Buffer Descriptors as shown in Figure 7.



D*a*... D*b*	Invalid Receive Buffer Descriptors. LSW Address contains all 1s.
D*i* ... D*j*	Valid Receive Buffer Descriptors that contain the locations of free Receive Buffers that the NIC may use to store received packets. Note that Receive Buffers are 4B aligned, so valid Descriptors cannot have all 1s.
D*e* ... D*f*	Invalid Receive Buffer Descriptors. LSW Address contains all 1s.

Figure 7: Receive Buffer Ring.

Both the head and tail pointers are initialized to the first entry D*a* of the ring. The NIC initializes the entire ring with all 1s to indicate that the ring is empty. The NIC keeps track of only the head pointer and infers the tail pointer by examining the content of Descriptors. The device driver keeps track of both the head and tail pointers.

The NIC advances its head pointer when it uses the Receive Buffer Descriptor (hence its corresponding Receive Buffer) pointed by the head pointer. So, the NIC always consumes Receive Buffers in the order specified in Receive Buffer Rings. The device driver advances its tail pointer when it allocates new Receive Buffers. It creates Receive Buffer Descriptors that correspond to the new Receive Buffers and then copies them to a Receive Buffer Ring in the NIC memory (see below for the ways of copying them). The device driver advances its head pointer when it processes received packets. It computes the number of Receive Buffers used for the given received packet and then advances the head pointer by the same number of entries as the Receive Buffers used for the received packet.

There are two ways for the device driver to copy Receive Buffer Descriptors to Receive Buffer Rings in the NIC memory.

1. The device driver may copy Descriptors directly to their corresponding entries in the NIC memory. To do so, the driver first computes the locations of the corresponding entries of the ring in the NIC memory from the offset obtained through MXGEFW_CMD_GET_SMALL_RX_OFFSET or MXGEFW_CMD_GET_BIG_RX_OFFSET. Then, the driver writes Descriptors to those locations through PIO.
   If this method is used, the first Descriptor must be written last.
2. Alternatively, the NIC may copy the Descriptors to the offset ETH_RECV_SMALL for Small Receive Buffer Ring or to the offset ETH_RECV_BIG for Big Receive Buffer Ring.

The NIC then places the new Descriptors into the correct entries in either Receive Buffer Ring. To use this method, the device driver must write 8 new Descriptors (64B) at a time. As with Send Ring, this method is intended for a system that allows the device driver to enable write-combining for PIO accesses. The device driver should use write-combining to ensure that all 64B are transferred to the NIC in a single PCI Express packet. Although the NIC is able to combine the payloads of smaller PCI Express packets into 64B of Descriptors, it generally leads to performance degradations. If the device driver cannot use write-combining, then it should use the first method to copy Descriptors.

6.5. Received Packet Descriptor

Received Packet Descriptor is a 4B data structure that contains information about a received packet. The device driver uses this information to process received packets. The Descriptor format is shown in Table 8.

Field Name	Offset (B)	Size (B)	Description
Checksum	0	2	16-bit one's complement sum of the entire packet except the first 14B.
Length	2	2	The length of the packet in bytes. 0 indicates that this Descriptor is invalid.

Table 8: Received Packet Descriptor.

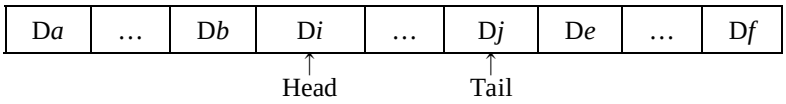
Using C code, this data structure may be declared as follows.

```
struct mcp_slot {
  uint16_t checksum;
  uint16_t length;
};
```

6.6. Interrupt Queue

Interrupt Queue is a ring of Received Packet Descriptors. The NIC produces new Received Packet Descriptors for received packets, and the device driver uses the information stored in Descriptors to process received packets. Interrupt Queue resides in main memory. During the initialization, the device driver allocates the ring and notifies the NIC of its address and size through MXGEFW_CMD_SET_INTRQ_DMA and MXGEFW_CMD_SET_INTRQ_SIZE. When the NIC transfers a received packet to Receive Buffer(s), the NIC creates a Received Packet Descriptor for the packet and then transfers it to Interrupt Queue in main memory.

Interrupt Queue must be one physically contiguous array of Received Packet Descriptors. Figure 8 illustrates Interrupt Queue.



D*a*... D*b*	Invalid Descriptors. Length in these Descriptors is 0.
D*i* ... D*j*	Valid Descriptors that have been produced by the NIC but have not yet been processed (consumed) by the device driver.
D*e* ... D*f*	Invalid Descriptors. Length in these Descriptors is 0.

Figure 8: Interrupt Queue.

The valid Descriptors that the device driver must process are surrounded by invalid Descriptors. Invalid Descriptors have Length equal to 0. Valid Descriptors must have non-zero Length. The head and tail pointers are both initialized to the first entry D*a*. The device driver initializes the entire ring with all 0s to indicate that the ring is empty. The NIC only keeps track of the tail pointer. When it produces a new Descriptor, it transfers it to the entry pointed by the tail pointer and advances the pointer to the next entry. The device driver only keeps track of the head pointer. The driver infers the tail pointer by checking Length. Thus, the driver first checks if Length of the Descriptor pointed by the head pointer is non-zero. If Length is non-zero, then the Descriptor is valid, and the driver processes it, sets Length to 0, and advances the head pointer to the next entry. The driver repeats this process until the Descriptor pointed by the head pointer contains Length 0.

The NIC assumes that whenever there is a free (either Small or Big) Receive Buffer, there is also a free entry in Interrupt Queue (i.e. the entry pointed by the tail pointer is invalid). The device driver may ensure this condition by setting the number of entries in Interrupt Queue to be at least the sum of the entries in Small Receive Buffer Ring and Big Receive Buffer Ring, and by allocating new Receive Buffers whenever an entry in Interrupt Queue is processed.

6.7. Operations

In order to receive packets, the device driver must first allocate Receive Buffers and produce Receive Buffer Descriptors in Receive Buffer Rings. The driver must allocate both Small and Big Receive Buffers.

The NIC uses the packet length to determine which buffer to use. If the packet length is at most

the size specified through MXGEFW_CMD_SET_SMALL_BUFFER_SIZE, the NIC uses a Small Receive Buffer. If the packet length is greater than the specified size, then the NIC uses one or more Big Receive Buffers. Once packets are stored in Receive Buffers, the NIC produces new Received Packet Descriptors and transfers them to the proper entries in Interrupt Queue in main memory and may generate an interrupt.

The device driver should follow the steps below in order to process received packets.

1. Examine the Received Packet Descriptor pointed by the head pointer of Interrupt Queue.
   If Length is non-zero, then the Descriptor is valid and contains the length and the IP checksum value of a received packet.
   If Length is zero, then there are no more received packets, and receive processing terminates.
2. Determine the Receive Buffer(s) that are used for storing the packet.
   If Length is at most the size specified through MXGEFW_CMD_SET_SMALL_BUFFER_SIZE, then the received packet is stored in the Small Receive Buffer that corresponds to the Receive Buffer Descriptor pointed by the head pointer of Small Receive Buffer Ring.
   If Length is greater than the size specified through MXGEFW_CMD_SET_SMALL_BUFFER_SIZE, then the received packet is stored in one or more Big Receive Buffers that correspond to the Receive Buffer Descriptors starting at the entry pointed by the head pointer of Big Receive Buffer Ring. Using Length in Received Packet Descriptor, the device driver must compute the number of Big Receive Buffers that are used to store the packet.
3. Perform operating system-specific processing.
   Checksum in Received Packet Descriptor contains the 16-bit one's complement sum of the entire packet minus the first 14 bytes. The exact use of this checksum value is specific to the operating system. In general, it is used to perform checksum offload.
4. Move the head pointer of Receive Buffer Rings.
   If a Small Receive Buffer is used for the packet, then advance the head pointer of Small Receive Buffer Ring by one entry.
   If one or more Big Receive Buffers are used for the packet, then advance the head pointer of Big Receive Buffer Ring by the same number of entries as the Big Receive Buffers.
5. Allocate new Receive Buffers.
   If one Small Receive Buffer is used for the packet, then allocate a new Small Receive Buffer. Create Receive Buffer Descriptor for the new Buffer and advance the tail pointer of Small Receive Buffer Ring by one entry.

If necessary, copy new Receive Buffer Descriptor(s) to Small Receive Buffer Ring in the NIC memory. See Section 6.4 for the ways to copy the Descriptors.

If one or more Big Receive Buffers are used for the packet, then allocate the same number of new Big Receive Buffers. Create Receive Buffer Descriptors for the new Buffers and advance the tail pointer of Big Receive Buffer Ring by the same number of entries as the new Descriptors. If necessary, copy new Receive Buffer Descriptor(s) to Big Receive Buffer Ring in the NIC memory. See Section 6.4 for the ways to copy the Descriptors.

6. Invalidate the current Received Packet Descriptor by writing 0s to Length.
7. Advance the head pointer of Interrupt Queue by one entry.
8. Go to step 1.

Note that Checksum is the checksum of the entire frame, except the first 14 bytes. If the Ethernet header is VLAN-tagged, and the driver wishes to compute the checksum of the payload, then the device driver must subtract the 4 bytes (bytes 14 through 18) from the checksum value stored in Received Packet Descriptor.

6.8. Block on Receive Buffer

The NIC first receives packets from the network into its internal receive buffer. The NIC then attempts to transfer the packets to Receive Buffers in main memory that have been posted by the device driver. If there are no available Receive Buffers for a packet, then the NIC can either drop the packet or delay receive processing until Receive Buffers are available. If the flow control mechanism is enabled through MXGEFW_ENABLE_FLOW_CONTROL, then the NIC does not drop received packets and simply delays receive processing until new Receive Buffers become available. If the flow control mechanism is disabled through MXGEFW_DISABLE_FLOW_CONTROL, then the NIC drops received packets for which there are no available Receive Buffers.

7. Interrupt

The NIC generates interrupts in order to notify the device driver of various events. Currently, there are three types of events on the NIC that trigger interrupts.

1. Sent packets. When packets are sent, the NIC updates Send Done Count in Interrupt Data and may generate an interrupt.
2. Received packets. When packets arrive at the NIC and are transferred to Receive Buffers, the NIC transfers Received Packet Descriptors to Interrupt Queue and then may generate an interrupt.
3. Changes in the state of the NIC or statistics. When the state of the NIC such as the link status changes, the NIC updates fields in Interrupt Data and then may generate an interrupt.

The NIC supports INTx (PCI Express emulation of legacy interrupt request lines) and MSI modes of interrupt. Enabling INTx or MSI is accomplished through PCI configuration and is specific to the operating system. Thus, it is up to the device driver to determine which mode is appropriate and then enable it using a method provided by the operating system.

7.1. Interrupt Coalescing

When there are pending events such as received packets, the NIC generates at most one interrupt per a fixed interval. This interval period in microseconds is stored in a 4B integer variable. During the initialization, the device driver must set this interval. It must use MXGEFW_CMD_GET_INTR_COAL_DELAY_OFFSET in order to determine the offset of the interval variable, and then set an appropriate value through PIO.

7.2. Interrupt Acknowledge

Interrupt Acknowledge consists of two 4B variables that reside on the NIC. The device driver acquires the offset of Interrupt Acknowledge through MXGEFW_CMD_GET_IRQ_ACK_OFFSET during the initialization. The NIC and the device driver use these variables as flags to enable and disable interrupt generation. The two variables are as follows.

1. Interrupt Acknowledge 0

   This flag is located at the offset returned by MXGEFW_CMD_GET_IRQ_ACK_OFFSET. The flag enables/disables interrupt generation due to newly received packets. The NIC initializes the flag to 3. 3 indicates that the NIC may generate an interrupt because the device driver is not processing received packets. When the NIC generates an interrupt because there are newly received packets, the flag becomes 2. 2 indicates that the NIC may not generate an interrupt to notify the device driver of newly received packets, because the driver is already processing received packets. When the device driver finishes

processing received packets, it must set the flag to 3 through PIO.

2. Interrupt Acknowledge 1

   This flags is located at the offset returned by MXGEFW_CMD_GET_IRQ_ACK_OFFSET plus 4.

   This flag enables/disables interrupt generation. The NIC initializes the flag to 3. 3 indicates that the NIC may generate an interrupt. When the NIC generates an interrupt, it sets the flag to 0. 0 indicates that the NIC may not generate an interrupt under any circumstances, because the device driver is currently processing interrupt. When the device driver finishes processing interrupt, it must set the flag to 3 through PIO.

The purpose of having a separate flag (Interrupt Acknowledge 0) for controlling interrupts due to received packets is primarily to support operating system mechanisms for processing received packets in a context other than the interrupt handler. For example, Linux NAPI processes received packets in a non-interrupt context by polling the NIC and with the interrupt disabled. While the interrupt is disabled, the NIC may continue to update Interrupt Queue as well as Send Done Count in Interrupt Data in main memory. The reason for doing so is to have the device driver to perform as much processing as possible per interrupt.

7.3. Interrupt Data

Interrupt Data is a data structure that contains information required for the device driver to process an interrupt. The NIC has Interrupt Data in the NIC memory, and the device driver also its own Interrupt Data in main memory. During the initialization, the device driver notifies the NIC of the location and the size of Interrupt Data in main memory through MXGEFW_CMD_SET_STATS_DMA_V2. During normal operations, the NIC updates its Interrupt Data, copies it to Interrupt Data in main memory (driver's copy), and then generates an interrupt. Interrupt Data contains the fields shown in Table 9. Interrupt Data may grow in the future. New fields must be added at the start of this structure (i.e. they will appear before Dropped PAUSE).

Field Name	Offset (B)	Size (B)	Description
Reserved	0	4	For future use.
Dropped PAUSE	4	4	The number of PAUSE frames received.
Dropped Unicast Filtered	8	4	The number of received packets dropped because their unicast destination MAC address do not match the NIC's MAC address.
Dropped Bad CRC32	12	4	The number of received packets dropped due to

			CRC32 errors.
Dropped Bad PHY	16	4	The number of received packets dropped due to PHY errors.
Dropped Multicast Filtered	20	4	The number of received packets dropped due to multicast filtering.
Send Done Count	24	4	The number of sent packets so far. The device driver uses this field to de-allocate sent packets.
Link Up	28	4	Link status. Initialized to 2. 2: Undetermined. 0: Link is down. 1: Link is up.
Dropped Link Overflow	32	4	The number of received packets dropped due to insufficient receive buffer space.
Dropped Link Error or Filtered	36	4	Sum of Dropped PAUSE, Dropped Bad CRC32, Dropped Bad PHY, and Dropped Unicast Filtered.
Dropped Runt	40	4	The number of received packets dropped because they are too small (smaller than 64B).
Dropped Overrun	44	4	The number of received packets dropped because they are too big (larger than MTU).
Dropped No Small Buffer	48	4	The number of received packets dropped because there are no Small Receive Buffers. When Block on Receive Buffer is enabled, NIC pauses receive instead of dropping the packet.
Dropped No Big Buffer	52	4	The number of received packets dropped because there are no Big Receive Buffers. When Block on Receive Buffer is enabled, NIC pauses receive instead of dropping the packet.
RDMA Tags Available	56	4	The number of usable read DMA tags on the NIC. It is normally 15.
TX Stopped	60	1	Non-zero value indicates that the transmit MAC stopped transmission due to the Ethernet flow control. This field must be located at a 4B-aligned address.
Link Down	61	1	Initialized to 0. 1 indicates that the NIC is ready to reset its state. This field is used for MXGEFW_CMD_ETHERNET_DOWN.
Stats Updated	62	1	Non-zero value indicates that some of the fields in Interrupt Data other than Send Done Count

			and Valid have been updated.
Valid	63	1	0 indicates that Interrupt Data has not been updated by the NIC. Bit 0 is set to indicate that new Received Packet Descriptors have been transferred to Interrupt Queue in the device driver. This bit is never set if Interrupt Acknowledge 0 is 2. Bit 1 is set to indicate that Interrupt Data has been updated by the NIC. See Section 7.4 for its use.

Table 9: Format of Interrupt Data.

Using C code, Interrupt Data may be declared as follows.

```
struct mcp_irq_data {
  /* add new counters at the beginning */
  uint32_t future_use[1];
  uint32_t dropped_pause;
  uint32_t dropped_unicast_filtered;
  uint32_t dropped_bad_crc32;
  uint32_t dropped_bad_phy;
  uint32_t dropped_multicast_filtered;
  /* 40 Bytes */
  uint32_t send_done_count;
  uint32_t link_up;
  uint32_t dropped_link_overflow;
  uint32_t dropped_link_error_or_filtered;
  uint32_t dropped_runt;
  uint32_t dropped_overrun;
  uint32_t dropped_no_small_buffer;
  uint32_t dropped_no_big_buffer;
  uint32_t rdma_tags_available;
  uint8_t tx_stopped;
  uint8_t link_down;
  uint8_t stats_updated;
  uint8_t valid;
};
```

7.4. Operations

INTx and MSI interrupt modes require slightly different actions from the device driver and the NIC.

7.4.1. Interrupt Processing Using INTx

Using INTx, the device driver should perform the following steps to process interrupts. Note that the driver is free to use any implementation as long as it provides the same functionality as the steps shown below.

1. Upon entering the interrupt handler, check Valid in Interrupt Data. If the value is zero, then Interrupt Data has not been updated by the NIC, and the device driver must exit the interrupt handler.
2. Save Valid.
3. Write any value to the offset obtained through MXGEFW_CMD_GET_IRQ_DEASSERT_OFFSET through PIO. This write triggers the NIC to de-assert the interrupt request and then to write 0 to Valid to indicate that the interrupt request has been de-asserted and that the driver may exit the interrupt handler.
4. Check bit 0 of the saved Valid. If the bit is set, then the NIC may have produced new Received Packet Descriptors and transferred them to Interrupt Queue (i.e. there are received packets to process). There are two ways to process received packets.
   A. If the operating system implements a mechanism for processing received packets in a non-interrupt context with the interrupt disabled, then the driver should take appropriate actions to start such mechanism.

      The driver should use the steps described in Section 6.7 to process received packets. Once received packets are processed, the driver must write 3 to Interrupt Acknowledge 0.

   B. Otherwise, process received packets using the steps described in Section 6.7.
5. Check bit 1 of the saved Valid. If the bit is set, then the NIC may have updated Send Done Count (i.e. packets have been sent). If the bit is set, process the sent packets using the steps described in Section 5.7. Note that currently, this bit is always set when the NIC generates an interrupt.
6. Check Valid. If the value is non-zero, then repeat steps from step 5. The device driver must check Valid before exiting the interrupt handler to make sure that the NIC has de-asserted the interrupt. Otherwise, the driver risks re-entering the handler immediately upon the exit because the interrupt request has not been de-asserted.
7. Check Stats Updated. If the value is non-zero, then the NIC may have updated statistics

variables and NIC state variables in Interrupt Data. For instance, the link state may have changed. If the value is non-zero, check all statistics and NIC state variables in Interrupt Data and take appropriate driver-specific actions.

8. Before exiting the interrupt handler, write 3 to Interrupt Acknowledge 1 to enable future interrupt generations.
9. If bit 0 of Valid was set, and the received packets were processed within the interrupt handler, then write 3 to Interrupt Acknowledge 0 in order to allow the NIC to generate interrupts when more packets are received.

7.4.2. Interrupt Processing Using MSI

Using MSI, the device driver should use the following steps to process interrupts. Again, the driver is free to use any implementation as long as it provides the same functionality as the steps shown below.

1. Upon entering the interrupt handler, check Valid in Interrupt Data. If the value is zero, then the device driver must exit the interrupt handler.
2. Check bit 0 of Valid. If the bit is set, then take one of the steps below, as appropriate.
   A. If the operating system implements a mechanism for processing received packets in a non-interrupt context with the interrupt disabled, then take appropriate actions to start such mechanism.

      Process received packets using the steps described in Section 6.7, and write 3 to Interrupt Acknowledge 0 once processing completes.

   B. Otherwise, process received packets using the steps described in Section 6.7.
3. Check bit 1 of Valid. If the bit is set, then process the sent packets using the steps described in Section 5.7.
4. Check Stats Updated. If the value is non-zero, then check all statistics and NIC state variables in Interrupt Data and take appropriate driver-specific actions.
5. Write 3 to Interrupt Acknowledge 1 to enable future interrupt generations.
6. If bit 0 of Valid was set, and the received packets were processed within the interrupt handler, then write 3 to Interrupt Acknowledge 0 in order to allow the NIC to generate interrupts when more packets are received.

8. Miscellaneous

8.1. Multicast filtering

If multicast filtering is enabled, then the NIC checks the destination MAC address of every received multicast Ethernet frame and determines if the address is allowed. If the address matches one of the multicast addresses in the multicast filtering table, then the NIC transfers the received packet to main memory. Otherwise, the received packet is dropped. The device driver may enable or disable multicast filtering at any time during normal operation.

The device driver must configure multicast filtering through the following ETH_CMD commands.

1. MXGEFW_ENABLE_ALLMULTI
2. MXGEFW_DISABLE_ALLMULTI
3. MXGEFW_JOIN_MULTICAST_GROUP
4. MXGEFW_LEAVE_MULTICAST_GROUP
5. MXGEFW_LEAVE_ALL_MULTICAST_GROUPS

8.2. mcp_gen_header

mcp_gen_header contains various types of information regarding the hardware and the currently running firmware. The structure has the fields shown in Table 10. This structure may grow in the future. New fields must be added to the bottom of the structure, below STRING_SPECS.

Field Name	Offset (B)	Size (B)	Description
Header Length	0	4	Size of mcp_gen_header in bytes.
MCP Type	4	4	Type of the running firmware. It has one of the values below. "MX": MX firmware. "PCIE": PCI Express-only firmware. "ETH": Ethernet firmware. "MCP0": MCP0 firmware.
Version String	8	128	Firmware version string.
MCP Globals	136	4	Address of structure specific to MCP Type.
SRAM Size	140	4	Size of SRAM.
STRING_SPECS Location	144	4	Address of STRING_SPECS.
STRING_SPECS Length	148	4	Size of STRING_SPECS in bytes.

Table 10: Format of mcp_gen_header.

Using C code, this data structure may be declared as follows.

```
struct mcp_gen_header {
  unsigned header_length;
  unsigned mcp_type;
  char version[128];
  unsigned mcp_globals;
  unsigned sram_size;
  unsigned string_specs;
  unsigned string_specs_len;
};
```

To access this structure, the device driver first reads 4B from offset 0x3C through PIO. The returned value is the offset of the structure in the NIC memory.

Version String contains information about the firmware and may look like the following.
"1.4.7 -H- 2006/10/11 22:26:24 PCIE-only firmware"
"1.4.7 -PRH- 2006/10/11 22:26:24 myri10ge firmware"

8.3. STRING_SPECS

STRING_SPECS is a string of ASCII characters containing information about the hardware. Its maximum length is 256B (256 characters). It consists of a number of "NAME=VALUE\0" strings. For example, STRING_SPECS may look like the following.

"MAC=00:60:dd:48:0b:db\0SN=268005\0PWR=100\0PC=10G-PCIE-8A-C\0PN=09-03177\0"

"\0" is the null character (value 0). MAC indicates the default MAC address. STRING_SPECS always includes the default MAC address. STRING_SPECS is currently located at fixed offset 0x1DFE00. Alternatively, the device driver may find its location by reading STRING_SPECS Location in mcp_gen_header.

8.4. PCI Power Management

The NIC supports PCI Power Management Capability and implements D0 and $D3_{hot}$ power states. The NIC can also report the estimated maximum power consumption. The device driver and/or the operating system control the state through PCI Configuration. For more details on how to use

Power Management Capability through PCI Configuration, refer to *PCI Bus Power Management Interface Specification*. The use of power management capability is entirely up to the device driver and the operating system.

Before changing the state to $D3_{hot}$, the device driver must ensure that the NIC will not initiate new DMA operations. For instance, the driver can achieve this effect by issuing MXGEFW_CMD_ETHERNET_DOWN (See Section 4.2) and also disabling BOOT_DUMMY_RDMA, before changing the power state to $D3_{hot}$. Depending on the driver implementation, transitioning from $D3_{hot}$ to D0 may require re-initialization of the NIC.

8.5. PCI Configuration

Table 11 shows the PCI configuration after a system reset. The values are expressed in little endian. For the exact definitions of the PCI configuration space and fields, refer to *PCI Local Bus Specification*, *PCI Express Base Specification*, and *PCI Bus Power Management Interface Specification*. Note that different versions of the NIC may use different configuration values.

Field Name	Offset (B)	Value	Sub-fields	Value	Notes
Vendor ID	0x00	0x14C1			
Device ID	0x02	0x0008			
Command	0x04	0x0000			
			I/O Space	0b	
			Memory Space	0b	
			Bus Master	0b	
			Special Cycles	0b	
			Memory Write and Invalidate Enable	0b	
			VGA Palette Snoop	0b	
			Parity Error Response	0b	
			SERR# Enable	0b	
			Fast Back-to-Back Enable	0b	
			Interrupt Disable	0b	
Status	0x06	0x0010			
			Interrupt Status	0b	
			Capabilities List	1b	
			66MHz Capable	0b	
			Fast Back-to-Back Capable	0b	

			Master Data Parity Error	0b	
			DEVSEL Timing	00b	
			Signaled Target Abort	0b	
			Received Target Abort	0b	
			Received Master Abort	0b	
			Signaled System Error	0b	
			Detected Parity Error	0b	
Revision ID	0x08	0x00			
Class Code	0x09	0x020000			Ethernet controller
Cacheline Size	0x0C	0x00			
Latency Timer	0x0D	0x00			
Header Type	0x0E	0x00			
BIST	0x0F	0x00			
			Completion Code	0000b	
			Start BIST	0b	
			BIST Capable	0b	
BAR0	0x10	0xXX00000C			
			Memory Space Indicator	0b	
			Type	10b	64-bit space
			Prefetchable	1b	
			Bits 4-23	0s	16MB
BAR0 Upper Address	0x14	0xXXXXXXXX			
BAR2	0x18	0xXXX00004			
			Memory Space Indicator	0b	64-bit space
			Type	10b	
			Prefetchable	0b	
			Bits 4-19	0s	1MB
BAR2 Upper Address	0x1C	0xXXXXXXXX			
Cardbus CIS Pointer	0x28	0x00000000			
Subsystem Vendor ID	0x2C	0x14C1			
Subsystem ID	0x2D	0x0008			
Expansion ROM Base Address	0x30	XXXX XXXX XXXX X000 0000 0000 0000 0000b			
			Expansion ROM Enable	0b	

			Bits 1-18	0s	512KB
Capabilities Pointer	0x34	0x44			
Interrupt Line	0x3C				Undefined
Interrupt Pin	0x3D	0x01			
Min_Gnt	0x3E	0x00			
Max_Lat	0x3F	0x00			
End of Configuration Space Header					
MSI Capability starts at 0x44					
Capability ID	0x00	0x05			
Next Pointer	0x01	0x54			
Message Control	0x02	0x0080			
			MSI Enable	0b	
			Multiple Message Capable	000b	1 message
			Multiple Message Enable	000b	
			& 64 bit address capable	1b	
			Per-vector masking capable	0b	
Message Address	0x04	0x00000000			
Message Upper Address	0x08	0x00000000			
Message Data	0x0C	0x0000			
Power Management Capability starts at 0x54					
Capability ID	0x00	0x01			
Next Pointer	0x01	0x5c			
PMC	0x02	0x0000			
			Version	010b	
			PME Clock	0b	
			DSI	0b	
			Aux_Current	000b	
			D1_Support	0b	
			D2_Support	0b	
			PME_Support	00000b	
PMCSR	0x04				
			PowerState	00b	
			PME_En	0b	

			Data_Select	0000b	
			Data_Scale	01b	0.1x
			PME_Status	0b	
Extentions	0x06		B2_B3#	0b	
			BPCC_En	0b	
Data	0x07	0x64			10W
PCI Express Capability starts at 0x5c					
Capability ID	0x00	0x10			
Next Pointer	0x01	0x88			
PCI Express Capabilities Register	0x02	0x0002			
			Capability Version	010b	
			Device/Port Type	0000b	PCI Express Endpoint device
			Slot Implemented	0b	
			Interrupt Message Number	00000b	
Device Capabilities	0x04	0x00000005			
			Max_Payload_Size Supported	101b	4096B
			Phantom Functions Supported	00b	
			Extended Tag Field Supported	0b	
			Endpoint L0s Acceptable Latency	000b	64ns
			Endpoint L1 Acceptable Latency	000b	1us
			Attention Button	0b	
			Attention Indicator	0b	
			Power Button	0b	
			Captured Slot Power Limit Value	00000000b	
			Captured Slot Power Limit Scale	00b	
Device Control	0x08	0x2810			
			Correctable Error Reporting Enable	0b	
			Non-Fatal Error Reporting Enable	0b	
			Fatal Error Reporting Enable	0b	
			Unsupported Request Error Reporting Enable	0b	

			Enable Relaxed Ordering	1b	
			Max_Payload_Size	000b	
			Extended Tag Field Enable	0b	
			Phantom Funtions Enable	0b	
			AUX Power PM Enable	0b	
			Enable No Snoop	1b	
			Max_Read_Request_Size	010b	
Device Status	0x0A	0x0000	Correctable Error Detected	0b	
			Non-Fatal Error Detected	0b	
			Fatal Error Detected	0b	
			Unsupported Request Detected	0b	
			AUX Power Detected	0b	
			Transactions Pending	0b	
Link Capabilities	0x0C	0x0003F481			
			Maximum Link Speed	0001b	
			Maximum Link Width	001000b	
			ASPM Support	01b	L0s
			L0s Exit Latency	111b	More than 4us
			L1 Exit Latency	111b	More than 64us
			Clock Power Management	0b	
			Surprise Down Error Reporting Capable	0b	
			Data Link Layer Link Active Reporting Capable	0b	
			Port Number	0000 0000b	
Link Control	0x10	0x0000			
			ASPM Control	00b	
			Read Completion Boundary	0b	
			Link Disable	0b	
			Retrain Link	0b	
			Common Clock Configuration	0b	
			Extended Synch	0b	
			Enable Clock Power Management	0b	
Link Status	0x12	0x0001			
			Link Speed	0001b	
			Negotiated Link Width	000000b	

			Training Error	0b	
			Link Training	0b	
			Slot Clock Configuration	0b	
Vendor Specific Capability starts at 0x88 Contact Myricom for information on this Capability					
Capability ID	0x00	0x09			
Next Pointer	0x01	0x00			End of Capabilities list
Capability Length	0x02	0x1C			
Reserved	0x04	0x00			
EEPROM Read Address	0x05	0x000000			
EEPROM Read Data	0x08	0x00000000			
EEPROM Write Data	0x0C	0x00			
EEPROM Write Address	0x0D	0x000000			
Debug Mode	0x10	0x00			
Debug Data	0x14	0x00000000			
Debug Address	0x18	0x00000000			
PCI Express Extended Capabilities start at 0x100					
Advanced Error Reporting Capability starts at 0x100					
Enhanced Capability Header	0x00	0x1A810001			
			Capability ID	0x00	0x0001
			Capability Version	0x1	
			Next Capability Offset	0x1A8	
Uncorrectable Error Status	0x04	0x00000000			
			Link Training Error	0b	
			Data Link Protocol Error Mask	0b	
			Poisoned TLP Status	0b	
			Flow Control Protocol Error Status	0b	
			Completion Timeout Status	0b	
			Completer Abort Status	0b	
			Unexpected Completion Status	0b	
			Receiver Overflow Status	0b	
			Malformed TLP Status	0b	

			ECRC Error Status	0b	
			Unsupported Request Error Status	0b	
Uncorrectable Error Mask	0x08	0x00000000			
			Link Training Error Mask	0b	
			Data Link Protocol Error Mask	0b	
			Poisoned TLP Mask	0b	
			Flow Control Protocol Error Mask	0b	
			Completion Timeout Mask	0b	
			Completer Abort Mask	0b	
			Unexpected Completion Mask	0b	
			Receiver Overflow Mask	0b	
			Malformed TLP Mask	0b	
			ECRC Error Mask	0b	
			Unsupported Request Error Mask	0b	
Uncorrectable Error Severity	0x0C	0x00062011			
			Link Training Error Severity	1b	
			Data Link Protocol Error Severity	1b	
			Poisoned TLP Severity	0b	
			Flow Control Protocol Error Severity	1b	
			Completion Timeout Error Severity	0b	
			Completer Abort Error Severity	0b	
			Unexpected Completion Error Severity	0b	
			Receiver Overflow Error Severity	1b	
			Malformed TLP Severity	1b	
			ECRC Error Severity	0b	
			Unsupported Request Error Severity	0b	
Correctable Error Status	0x10	0x00000000			
			Receiver Error Status	0b	
			Bad TLP Status	0b	
			Bad DLLP Status	0b	
			REPLAY_NUM Rollover Status	0b	
			Replay Timer Timeout Status	0b	
Correctable Error Mask	0x14	0x00000000			
			Receiver Error Mask	0b	
			Bad TLP Mask	0b	

			Bad DLLP Mask	0b	
			REPLAY_NUM Rollover Mask	0b	
			Replay Timer Timeout Mask	0b	
Advanced Error Capabilities and Error Control	0x18	0x000000A0			
			First Error Pointer	00000b	
			ECRC Generation Capable	1b	
			ECRC Generation Enable	0b	
			ECRC Check Capable	1b	
			ECRC Check Enable	0b	
Header Log Register	0x1C	0x00000000			
Header Log Register	0x20	0x00000000			
Header Log Register	0x24	0x00000000			
Header Log Register	0x28	0x00000000			
Root Error Command	0x2C	0x00000000			Unused
Root Error Status	0x30	0x00000000			Unused
CESIR	0x34	0x0000			Unused
ESIR	0x36	0x0000			Unused
Device Serial Number Capability starts at 0x1A8					
Enhanced Capability Header	0x00	0x00010003			
			Capability ID	0x0003	
			Capability Version	0x1	
			Next Capability Offset	0x000	End of Capabilities list
Device Serial Number	0x04	0x00000000			Set to MAC address
Device Serial Number	0x08	0x00000000			Set to MAC address

Table 11: PCI configuration space after a reset.