Chapter 5 - NetX Duo Network Drivers

Contents

Chapter 5 - NetX Duo Network Drivers **2**
Driver Introduction . . . 2
Driver Entry . . . 3
Driver Requests . . . 3
Driver Initialization . . . 3
Enable Link . . . 4
Disable Link . . . 5
Uninitialize Link . . . 5
Packet Send . . . 6
Packet Broadcast(IPv4 packets only) . . . 7
ARP Send . . . 7
ARP Response Send . . . 8
RARP Send . . . 8
Multicast Group Join . . . 9
Multicast Group Leave . . . 9
Attach Interface . . . 10
Detach Interface . . . 10
Get Link Status . . . 11
Get Link Speed . . . 11
Get Duplex Type . . . 12
Get Error Count . . . 12
Get Receive Packet Count . . . 13
Get Transmit Packet Count . . . 13
Get Allocation Errors . . . 13
Driver Deferred Processing . . . 14
Set Physical Address . . . 14
User Commands . . . 15
Unimplemented Commands . . . 15
Driver Capability . . . 15
Driver Output . . . 16
Driver Input . . . 17
Deferred Receive Packet Handling . . . 18
Ethernet Headers . . . 18

Example RAM Ethernet Network Driver 19
TCP/IP Offload Driver Guidance 20
TCP/IP Offload Driver Thread 21
TCP/IP Offload Driver Handler 21
TSN driver support 22
PTP initialize and callback function 22
Credit-based shaper (CBS) - IEEE 802.1Qav Forwarding and Queuing Enhancements for Time-Sensitive Stream 22
Time-Aware Shaper (TAS) - IEEE 802.1Qbv Enhancements to Traffic Scheduling 23
Frame preemption (FPE) - 802.1Qbu 24

Chapter 5 - NetX Duo Network Drivers

This chapter contains a description of network drivers for NetX Duo. The information presented is designed to help developers write application-specific network drivers for NetX Duo.

Driver Introduction

The NX_IP structure contains everything to manage a single IP instance. This includes general TCP/IP protocol information as well as the application-specific physical network driver's entry routine. The driver's entry routine is defined during the ***nx_ip_create*** service. Additional devices may be added to the IP instance via the ***nx_ip_interface_attach*** service.

Communication between NetX Duo and the application's network driver is accomplished through the **NX_IP_DRIVER** request structure. This structure is most often defined locally on the caller's stack and is therefore released after the driver and calling function return. The structure is defined as follows.

```
typedef struct NX_IP_DRIVER_STRUCT
{
      UINT           nx_ip_driver_command;
      UINT           nx_ip_driver_status;
      ULONG          nx_ip_driver_physical_address_msw;
      ULONG          nx_ip_driver_physical_address_lsw;
      NX_PACKET      *nx_ip_driver_packet;
      ULONG          *nx_ip_driver_return_ptr;
      NX_IP          *nx_ip_driver_ptr;
      NX_INTERFACE   *nx_ip_driver_interface;01
} NX_IP_DRIVER;
```

Driver Entry

NetX Duo invokes the network driver entry function for driver initialization and for sending packets and for various control and status operations, including initializing and enabling the network device. NetX Duo issues commands to the network driver by setting the ***nx_ip_driver_command*** field in the **NX_IP_DRIVER** request structure. The driver entry function has the following format:

```
VOID my_driver_entry(NX_IP_DRIVER *request);
```

Driver Requests

NetX Duo creates the driver request with a specific command and invokes the driver entry function to execute the command. Because each network driver has a single entry function, NetX Duo makes all requests through the driver request data structure. The ***nx_ip_driver_command*** member of the driver request data structure (**NX_IP_DRIVER**) defines the request. Status information is reported back to the caller in the member ***nx_ip_driver_status***. If this field is **NX_SUCCESS**, the driver request was completed successfully.

NetX Duo serializes all access to the driver. Therefore, the driver does not need to handle multiple threads asynchronously calling the entry function. Note that the device driver function executes with the IP mutex locked. Therefore the device driver internal function shall not block itself.

Typically the device driver also handles interrupts. Therefore, all driver functions need to be interrupt-safe.

Driver Initialization

Although the actual driver initialization processing is application specific, it usually consists of data structure and physical hardware initialization. The information required from NetX Duo for driver initialization is the IP Maximum Transmission Unit (MTU), which is the number of bytes available to the IP-layer payload, including IPv4 or IPv6 header) and if the physical interface needs logical-to-physical mapping. The driver configures the interface MTU value by calling ***nx_ip_interface_mtu_set***.

The device driver needs to call ***nx_ip_interface_address_mapping_configure*** to inform NetX Duo whether or not interface address mapping is required. If address mapping is needed, the driver is responsible for configuring the interface with valid MAC address, and supply the MAC address to NetX via ***nx_ip_interface_physical_address_set***.

When the network driver receives the NX_LINK INITIALIZE request from NetX Duo, it receives a pointer to the IP control block as part of the NX_IP_DRIVER request control block shown above.

After the application calls ***nx_ip_create***, the IP helper thread sends a driver request with the command set to NX_LINK_INITIALIZE to the driver to initialize its physical network interface. The following NX_IP_DRIVER members are used for the initialize request.

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_INITIALIZE
nx_ip_driver_ptr	Pointer to the IP instance. This value should be saved by the driver so that the driver function can find the IP instance to operate on.
nx_ip_driver_interface	Pointer to the network interface structure within the IP instance. This information should be saved by the driver. On receiving packets, the driver shall use the interface structure information when sending the packet up the stack. The interface index (device index) can be obtained by reading the member nx_interface_index inside this data structure.
nx_ip_driver_status	Completion status. If the driver is not able to initialize the specified interface to the IP instance, it will return a nonzero error status.

> **Note:** *The driver is actually called from the IP helper thread that was created for the IP instance. Therefore the driver routine should avoid performing blocking operations, or the IP helper thread could stall, causing unbounded delays to applications that rely on the IP thread.*

Enable Link

Next, the IP helper thread enables the physical network by setting the driver command to NX_LINK_ENABLE in the driver request and sending the request to the network driver. This happens shortly after the IP helper thread completes the initialization request. Enabling the link may be as simple as setting the *nx_interface_link_up* field in the interface instance. But it may also involve manipulation of the physical hardware. The following NX_IP_DRIVER members are used for the enable link request.

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_ENABLE

NX_IP_DRIVER member	Meaning
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_interface	Pointer to the interface instance
nx_ip_driver_status	Completion status. If the driver is not able to enable the specified interface, it will return a non-zero error status.

Disable Link

This request is made by NetX Duo during the deletion of an IP instance by the ***nx_ip_delete*** service. Or an application may issue this command in order to temporarily disable the link in order to save power. This service disables the physical network interface on the IP instance. The processing to disable the link may be as simple as clearing the *nx_interface_link_up* flag in the interface instance. But it may also involve manipulation of the physical hardware. Typically it is a reverse operation of the ***Enable Link*** operation. After the link is disabled, the application request ***Enable Link*** operation to enable the interface.

The following NX_IP_DRIVER members are used for the disable link request.

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_DISABLE
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_interface	Pointer to the interface instance
nx_ip_driver_status	Completion status. If the driver is not able to disable the specified interface in the IP instance, it will return a non-zero error status.

Uninitialize Link

This request is made by NetX Duo during the deletion of an IP instance by the ***nx_ip_delete*** service. This request uninitialized the interface, and release any resources created during initialization phase. Typically it is a reverse operation of the ***Initialize Link*** operation. After the interface is uninitialized, the device cannot be used until the interface is initialized again.

The following NX_IP_DRIVER members are used for the disable link request.

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_UNINITIALIZE
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_interface	Pointer to the interface instance

NX_IP_DRIVER member	Meaning
nx_ip_driver_status	Completion status. If the driver is not able to uninitialize the specified interface to the IP instance, it will return a non-zero error status.

Packet Send

This request is made during internal IPv4 or IPv6 send processing, which all NetX Duo protocols use to transmit packets (except for ARP, RARP). On receiving the packet send command, the *nx_packet_prepend_ptr* points to the beginning of the packet to be sent, which is the beginning of the IPv4 or IPv6 header. *nx_packet_length* indicates the total size (in bytes) of the data being transmitted. If *nx_packet_next* is valid, the outgoing IP datagram is stored in multiple packets, the driver is required to follow the chained packet and transmit the entire frame. Note that valid data area in each chained packet is stored between *nx_packet_prepend_ptr* and *nx_packet_append_ptr*.

The driver is responsible for constructing physical header. If physical address to IP address mapping is required (such as Ethernet), the IP layer already resolved the MAC address. The destination MAC address is passed from the IP instance, stored in *nx_ip_driver_physical_address_msw and nx_ip_driver_physical_address_lsw*.

After adding the physical header, the packet send processing then calls the driver's output function to transmit the packet.

The following NX_IP_DRIVER members are used for the packet send request.

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_PACKET_SEND
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_packet	Pointer to the packet to send
nx_ip_driver_interface	Pointer to the interface instance.
nx_ip_driver_physical_address_msw	Most significant 32-bits of physical address (only if physical mapping needed)
nx_ip_driver_physical_address_lsw	Least significant 32-bits of physical address (only if physical mapping needed)
nx_ip_driver_status	Completion status. If the driver is not able to send the packet, it will return a non-zero error status.

Packet Broadcast(IPv4 packets only)

This request is almost identical to the send packet request. The only difference is that the destination physical address fields are set to the Ethernet broadcast MAC address. The following NX_IP_DRIVER members are used for the packet broadcast request.

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_PACKET_BROADCAST
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_packet	Pointer to the packet to send
nx_ip_driver_physical_address_msw	0x0000FFFF (broadcast)
nx_ip_driver_physical_address_lsw	0xFFFFFFFF (broadcast)
nx_ip_driver_interface	Pointer to the interface instance.
nx_ip_driver_status	Completion status. If the driver is not able to send the packet, it will return a non-zero error status.

ARP Send

This request is also similar to the IP packet send request. The only difference is that the Ethernet header specifies an ARP packet instead of an IP packet, and destination physical address fields are set to MAC broadcast address. The following NX_IP_DRIVER members are used for the ARP send request.

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_ARP_SEND
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_packet	Pointer to the packet to send
nx_ip_driver_physical_address_msw	0x0000FFFF (broadcast)
nx_ip_driver_physical_address_lsw	0xFFFFFFFF (broadcast)
nx_ip_driver_interface	Pointer to the interface instance.
nx_ip_driver_status	Completion status. If the driver is not able to send the ARP packet, it will return a non-zero error status.

> **Important:** *If physical mapping is not needed, implementation of this request is not required.*

Although ARP has been replaced with the Neighbor Discovery Protocol and the Router Discovery Protocol in IPv6, Ethernet network drivers must still be compatible with IPv4 peers and routers. Therefore, drivers must still handle ARP packets.

ARP Response Send

This request is almost identical to the ARP send packet request. The only difference is the destination physical address fields are passed from the IP instance. The following NX_IP_DRIVER members are used for the ARP response send request.

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_ARP_RESPONSE_SEND
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_packet	Pointer to the packet to send
nx_ip_driver_physical_address_msw	Most significant 32-bits of physical address
nx_ip_driver_physical_address_lsw	Least significant 32-bits of physical address
nx_ip_driver_interface	Pointer to the interface instance
nx_ip_driver_status	Completion status. If the driver is not able to send the ARP packet, it will return a non-zero error status.

> **Important:** *If physical mapping is not needed, implementation of this request is not required.*

RARP Send

This request is almost identical to the ARP send packet request. The only differences are the type of packet header and the physical address fields are not required because the physical destination is always a broadcast address.

The following NX_IP_DRIVER members are used for the RARP send request.

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_RARP_SEND
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_packet	Pointer to the packet to send
nx_ip_driver_physical_address_msw	0x0000FFFF (broadcast)
nx_ip_driver_physical_address_lsw	0xFFFFFFFF (broadcast)
nx_ip_driver_interface	Pointer to the interface instance.
nx_ip_driver_status	Completion status. If the driver is not able to send the RARP packet, it will return a non-zero error status.

> **Important:** *Applications that require RARP service must implement this command.*

Multicast Group Join

This request is made with the ***nx_igmp_multicast_interface join*** and ***nx_ipv4_multicast_interface_join*** service in IPv4, ***nxd_ipv6_multicast_interface_join*** service in IPv6, and various operation required by IPv6. The network driver takes the supplied multicast group address and sets up the physical media to accept incoming packets from that multicast group address. Note that for drivers that don't support multicast filter, the driver receive logic may have to be in promiscuous mode. In this case, the driver may need to filter incoming frames based on destination MAC address, thus reducing the amount of traffic passed into the IP instance. The following NX_IP_DRIVER members are used for the multicast group join request.

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_MULTICAST_JOIN
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_physical_address_msw	Most significant 32-bits of physical multicast address
nx_ip_driver_physical_address_lsw	Least significant 32-bits of physical multicast address
nx_ip_driver_interface	Pointer to the interface instance
nx_ip_driver_status	Completion status. If the driver is not able to join the multicast group, it will return a non-zero error status.

Note: *IPv6 applications will require multicast to be implemented in the driver for ICMPv6 based protocols such as address configuration. However, for IPv4 applications, implementation of this request is not necessary if multicast capabilities are not required.*

Important: *If IPv6 is not enabled, and multicast capabilities are not required by IPv4, implementation of this request is not required.*

Multicast Group Leave

This request is invoked by explicitly calling the ***nx_igmp_multicast_interface_leave*** or ***nx_ipv4_multicast_interface_leave*** services in IPv4, ***nxd_ipv6_multicast_interface_leave*** service in IPv6, or by various internal NetX Duo operations required for IPv6. The driver removes the supplied Ethernet multicast address from the multicast list. After a host has left a multicast group, packets on the network with this Ethernet multicast address are no longer received by this IP instance. The following NX_IP_DRIVER members are used for the multicast group leave request.

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_MULTICAST_LEAVE
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_physical_address_msw	Most significant 32 bits of physical multicast address
nx_ip_driver_physical_address_lsw	Least significant 32 bits of physical multicast address
nx_ip_driver_interface	Pointer to the interface instance
nx_ip_driver_status	Completion status. If the driver is not able to leave the multicast group, it will return a non-zero error status.

> **Important:** *If multicast capabilities are not required by either IPv4 or IPv6, implementation of this request is not required.*

Attach Interface

This request is invoked from the NetX Duo to the device driver, allowing the driver to associate the driver instance with the corresponding IP instance and the physical interface instance within the IP. The following NX_IP_DRIVER members are used for the attach interface request.

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_INTERFACE_ATTACH
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_interface	Pointer to the interface instance.
nx_ip_driver_status	Completion status. If the driver is not able to detach the specified interface to the IP instance, it will return a non-zero error status.

Detach Interface

This request is invoked by NetX Duo to the device driver, allowing the driver to disassociate the driver instance with the corresponding IP instance and the physical interface instance within the IP. The following NX_IP_DRIVER members are used for the attach interface request.

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_INTERFACE_DETACH
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_interface	Pointer to the interface instance.

NX_IP_DRIVER member	Meaning
nx_ip_driver_status	Completion status. If the driver is not able to attach the specified interface to the IP instance, it will return a non-zero error status.

Get Link Status

The application can query the network interface link status using the NetX Duo service ***nx_ip_interface_status_check*** service for any interface on the host. See Chapter 4, "Description of NetX Duo Services" on page 149, for more details on these services.

The link status is contained in the *nx_interface_link_up* field in the NX_INTERFACE structure pointed to by *nx_ip_driver_interface* pointer. The following NX_IP_DRIVER members are used for the link status request.

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_GET_STATUS
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_return_ptr	Pointer to the destination to place the status.
nx_ip_driver_interface	Pointer to the interface instance
nx_ip_driver_status	Completion status. If the driver is not able to get specific status, it will return a non-zero error status.

> **Note:** ***nx_ip_status_check*** is still available for checking the status of the primary interface. However, application developers are encouraged to use the interface specific service: ***nx_ip_interface_status_check.***

Get Link Speed

This request is made from within the ***nx_ip_driver_direct_command*** service. The driver stores the link's line speed in the supplied destination. The following NX_IP_DRIVER members are used for the link line speed request.

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_GET_SPEED
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_return_ptr	Pointer to the destination to place the line speed
nx_ip_driver_interface	Pointer to the interface instance

NX_IP_DRIVER member	Meaning
nx_ip_driver_status	Completion status. If the driver is not able to get speed information, it will return a non-zero error status.

Important: *This request is not used internally by NetX Duo so its implementation is optional.*

Get Duplex Type

This request is made from within the ***nx_ip_driver_direct_command*** service. The driver stores the link's duplex type in the supplied destination. The following NX_IP_DRIVER members are used for the duplex type request.

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_GET_DUPLEX_TYPE
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_return_ptr	Pointer to the destination to place the duplex type
nx_ip_driver_interface	Pointer to the interface instance
nx_ip_driver_status	Completion status. If the driver is not able to get duplex information, it will return a nonzero error status.

Important: *This request is not used internally by NetX Duo so its implementation is optional.*

Get Error Count

This request is made from within the ***nx_ip_driver_direct_command*** service. The driver stores the link's error count in the supplied destination. To support this feature, the driver needs to track operation errors. The following NX_IP_DRIVER members are used for the link error count request.

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_GET_ERROR_COUNT
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_return_ptr	Pointer to the destination to place the error count
nx_ip_driver_interface	Pointer to the interface instance
nx_ip_driver_status	Completion status. If the driver is not able to get error count, it will return a non-zero error status.

Important: *This request is not used internally by NetX Duo so its implementation is optional.*

Get Receive Packet Count

This request is made from within the ***nx_ip_driver_direct_command*** service. The driver stores the link's receive packet count in the supplied destination. To support this feature, the driver needs to keep track of the number of packets received. The following NX_IP_DRIVER members are used for the link receive packet count request.

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_GET_RX_COUNT
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_return_ptr	Pointer to the destination to place the receive packet count
nx_ip_driver_interface	Pointer to the physical network interface
nx_ip_driver_status	Completion status. If the driver is not able to get receive count, it will return a non-zero error status.

> **Important:** *This request is not used internally by NetX Duo so its implementation is optional.*

Get Transmit Packet Count

This request is made from within the ***nx_ip_driver_direct_command*** service. The driver stores the link's transmit packet count in the supplied destination. To support this feature, the driver needs to keep track of each packet it transmits on each interface. The following NX_IP_DRIVER members are used for the link transmit packet count request.

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_GET_TX_COUNT
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_return_ptr	Pointer to the destination to place the transmit packet count
nx_ip_driver_interface	Pointer to the interface instance
nx_ip_driver_status	Completion status. If the driver is not able to get transmit count, it will return a non-zero error status.

> **Important:** *This request is not used internally by NetX Duo so its implementation is optional.*

Get Allocation Errors

This request is made from within the ***nx_ip_driver_direct_command*** service. The driver stores the link's packet pool allocation error count in the supplied destination. The following NX_IP_DRIVER members are used for the link allocation error count request.

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_GET_ALLOC_ERRORS
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_return_ptr	Pointer to the destination to place the allocation error count
nx_ip_driver_interface	Pointer to the interface instance
nx_ip_driver_status	Completion status. If the driver is not able to get allocation errors, it will return a non-zero error status.

> **Important:** *This request is not used internally by NetX Duo so its implementation is optional.*

Driver Deferred Processing

This request is made from the IP helper thread in response to the driver calling the ***_nx_ip_driver_deferred_processing*** routine from a transmit or receive ISR. This allows the driver ISR to defer the packet receive and transmit processing to the IP helper thread and thus reduce the amount to process in the ISR. The *nx_interface_additional_link_info* field in the NX_INTERFACE structure pointed to by *nx_ip_driver_interface* may be used by the driver to store information about the deferred processing event from the IP helper thread context. The following NX_IP_DRIVER members are used for the deferred processing event.

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_DEFERRED_PROCESSING
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_interface	Pointer to the interface instance

Set Physical Address

This request is made from within the ***nx_ip_interface_physical_address_set*** service. This service allows an application to change the interface physical address at run time. On receiving this command, the driver is required to re-configure the hardware address of the network interface to the supplied physical address. Since the IP instance already has the new address, there is no need to call the ***nx_ip_interface_address_set*** service from this command.

The following NX_IP_DRIVER members are used for the user command request.

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_SET_PHYSICAL_ADDRESS
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_interface	Pointer to the interface instance
nx_ip_driver_physical_address_msw	Most significant 32-bits of the new physical address
nx_ip_driver_physical_address_lsw	Least significant 32-bits of the new physical address
nx_ip_driver_status	Completion status. If the driver is not able to reconfigure the physical address, it will return a non-zero error status.

User Commands

This request is made from within the ***nx_ip_driver_direct_command*** service. The driver processes the application specific user commands. The following NX_IP_DRIVER members are used for the user command request.

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_USER_COMMAND
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_return_ptr	User defined
nx_ip_driver_interface	Pointer to the interface instance
nx_ip_driver_status	Completion status. If the driver is not able to execute user commands, it will return a non-zero error status.

Important: *This request is not used internally by NetX Duo so its implementation is optional.*

Unimplemented Commands

Commands unimplemented by the network driver must have the return status field set to NX_UNHANDLED_COMMAND.

Driver Capability

Some network interfaces offer checksum offload features. Device drivers may take advantage of the hardware accelerations to free up CPU time from running various checksum computations.

Depending the level of hardware checksum support from the hardware, the device driver needs to inform the IP instance which hardware feature is enabled. This way, the IP instance is aware of the hardware feature, and offload as much computation to the hardware as possible. The driver should use the API

nx_ip_interface_capability_set to set all the features the physical interface is able to handle.

The following features can be used:

- NX_INTERFACE_CAPABILITY_IPV4_TX_CHECKSUM
- NX_INTERFACE_CAPABILITY_IPV4_RX_CHECKSUM
- NX_INTERFACE_CAPABILITY_TCP_TX_CHECKSUM
- NX_INTERFACE_CAPABILITY_TCP_RX_CHECKSUM
- NX_INTERFACE_CAPABILITY_UDP_TX_CHECKSUM
- NX_INTERFACE_CAPABILITY_UDP_RX_CHECKSUM
- NX_INTERFACE_CAPABILITY_ICMPV4_TX_CHECKSUM
- NX_INTERFACE_CAPABILITY_ICMPV4_RX_CHECKSUM
- NX_INTERFACE_CAPABILITY_ICMPV6_TX_CHECKSUM
- NX_INTERFACE_CAPABILITY_ICMPV6_RX_CHECKSUM
- NX_INTERFACE_CAPABILITY_IGMP_TX_CHECKSUM
- NX_INTERFACE_CAPABILITY_IGMP_RX_CHECKSUM

For a checksum computation that can be performed in hardware, the driver must set up the hardware or the buffer descriptors correctly so the checksum for an out-going packet can be generated and inserted into the header by the hardware. On receiving a packet, the hardware checksum logic should be able to verify the checksum value. If the checksum value is incorrect, the received frame should be discarded.

Even with the capability of performing checksum computation in hardware, the IP instance still maintains the checksum capability. In certain scenarios, for example a UDP datagram going through IPsec protection, the UDP checksum must be computed in software before passing the UDP frame down the stack. Most hardware checksum feature does not support checksum computation for a segment of data protected by IPsec. For a UDP or ICMP frame that needs to be fragmented, the UDP or ICMP checksum needs to be computed in software. Most hardware checksum logic does not handle the case where the data is split into multiple frames.

Driver Output

All previously mentioned packet transmit requests require an output function implemented in the driver. Specific transmit logic is hardware specific, but it usually consists of checking for hardware capacity to send the packet immediately. If possible, the packet payload (and additional payloads in the packet chain) are loaded into one or more of the hardware transmit buffers and a transmit operation is initiated. If the packet won't fit in the available transmit buffers, the packet should be queued, and be transmitted when the transmission buffers become available.

The recommended transmit queue is a singly linked list, having both head and tail pointers. New packets are added to the end of the queue, keeping the oldest packet at the front. The *nx_packet_queue_next* field is used as the packet's next link in the queue. The driver defines the head and tail pointers of the transmit queue.

> **Caution:** *Because this queue is accessed from thread and interrupt portions of the driver, interrupt protection must be placed around the queue manipulations.*

Most physical hardware implementations generate an interrupt upon packet transmit completion. When the driver receives such an interrupt, it typically releases the resources associated with the packet just being transmitted. In case the transmit logic reads data directly from the NX_PACKET buffer, the driver should use the ***nx_packet_transmit_release*** service to release the packet associated with the transmit complete interrupt back to the available packet pool. Next, the driver examines the transmit queue for additional packets waiting to be sent. As many of the queued transmit packets that fit into the hardware transmit buffer(s) are de-queued and loaded into the buffers. This is followed by initiation of another send operation.

As soon as the data in the NX_PACKET has been moved into the transmitter FIFO (or in case a driver supports zero-copy operation, the data in NX_PACKET has been transmitted), the driver must move the *nx_packet_prepend_ptr* to the beginning of the IP header before calling ***nx_packet_transmit_release.*** Remember to adjust *nx_packet_length* field accordingly. If an IP frame is made up of multiple packets, only the head of the packet chain needs to be released.

Driver Input

Upon reception of a received packet interrupt, the network driver retrieves the packet from the physical hardware receive buffers and builds a valid NetX Duo packet. Building a valid NetX Duo packet involves setting up the appropriate length field and chaining together multiple packets if the incoming packet's size is greater than a single packet payload. Once properly built, the *prepend_ptr* is moved after the physical layer header and the receive packet is dispatched to NetX Duo.

NetX Duo assumes that the IP (IPv4 and IPv6) and ARP headers are aligned on a **ULONG** boundary. The NetX Duo driver must, therefore, ensure this alignment. In Ethernet environments this is done by starting the Ethernet header two bytes from the beginning of the packet. When the *nx_packet_prepend_ptr* is moved beyond the Ethernet header, the underlying IP (IPv4 and IPv6) or ARP header is 4-byte aligned.

> **Warning:** *See the section "Ethernet Headers" below for important differences between IPv6 and IPv6 Ethernet headers.*

There are several receive packet functions available in NetX Duo. If the received packet is an ARP packet, **nx_arp_packet_deferred_receive** *is called. If the received packet is an RARP packet,* ***nx_rarp_packet_deferred_receive*** is called. There are several options for handling incoming IP packets. For the fastest handling of IP packets, **nx__ip__packet__receive** *is called. This approach has the least overhead, but requires more processing in the driver's receive interrupt service handler (ISR). For minimal ISR processing* ***nx__ip__packet__deferred__receive*** is called.

After the new receive packet is properly built, the physical hardware's receive buffers are setup to receive more data. This might require allocating NetX Duo packets and placing the payload address in the hardware receive buffer or it may simply amount to changing a setting in the hardware receive buffer. To minimize overrun possibilities, it is important that the hardware's receive buffers have available buffers as soon as possible after a packet is received.

> **Important:** *The initial receive buffers are setup during driver initialization.*

Deferred Receive Packet Handling

The driver may defer receive packet processing to the NetX Duo IP helper thread. For some applications this may be necessary to minimize ISR processing as well as dropped packets.

To use deferred packet handling, the NetX Duo library must first be compiled with ***NX__DRIVER__DEFERRED__PROCESSING*** defined. This adds the deferred packet logic to the NetX Duo IP helper thread. Next, on receiving a data packet, the driver must call *_nx_ip_packet_deferred_receive():*

```
_nx_ip_packet_deferred_receive(ip_ptr, packet_ptr);
```

The deferred receive function places the receive packet represented by *packet_ptr* on a FIFO (linked list) and notifies the IP helper thread. After executing, the IP helper repetitively calls the deferred handling function to process each deferred packet. The deferred handler processing typically includes removing the packet's physical layer header (usually Ethernet) and dispatching it to one of these NetX Duo receive functions:

- ***_nx_ip_packet_receive***
- ***_nx_arp_packet_deferred_receive***
- ***_nx_rarp_packet_deferred_receive***

Ethernet Headers

One of the most significant differences between IPv6 and IPv4 for Ethernet Headers is the frame type setting. When sending out packets, the Ethernet driver is responsible for setting the Ethernet frame type in outgoing packets. For IPv6 packets, the frame type should be 0x86DD; for IPv4 packets, the frame type should be 0x0800.

The following code segment illustrates this process:

```
NX_PACKET *packet_ptr;
packet_ptr = driver_req_ptr -> nx_ip_driver_packet;
if (packet_ptr -> nx_packet_ip_version == NX_IP_VERSION_V4)
{

   /* Set Ethernet frame type to IPv4 /*
   ethernet_frame_ptr -> frame_type = 0x0800;

   /* Swap endian-ness for little endian targets.*/
   NX_CHANGE_USHORT_ENDIAN(ethernet_frame_ptr -> frame_type);
}
else if (packet_ptr -> nx_packet_ip_version == NX_IP_VERSION_V6)
{

   /* Set Ethernet frame type to IPv6. /*
   ethernet_frame_ptr -> frame_type = 0x86DD;

   /* Swap endian-ness for little endian targets.*/
   NX_CHANGE_USHORT_ENDIAN(ethernet_frame_ptr -> frame_type);
}
else
{

   /* Unknown IP version. Free the packet we will not send. */
   nx_packet_transmit_release(packet_ptr);
}
```

Similarly, for incoming packets, the Ethernet driver should determine the packet type from the Ethernet frame type. It should be implemented to accept IPv6 (0x86DD), IPv4 (0x0800), ARP (0x0806), and RARP (0x8035) frame types.

Example RAM Ethernet Network Driver

The NetX Duo demonstration system is delivered with a small RAM-based network driver, defined in the file ***nx_ram_network_driver.c.*** This driver assumes the IP instances are all on the same network and simply assigns virtual hardware addresses (MAC addresses) to each device instance as they are created. This file provides a good example of the basic structure of NetX Duo physical network drivers. Users may develop their own network drivers using the driver framework presented in this example.

The entry function of the network driver is ***_nx_ram_network_driver(),*** which is passed to the IP instance create call. Entry functions for additional network interfaces can be passed into the *nx_ip_interface_attach()* service. After the IP instance starts to run, the driver entry function is invoked to initialize and enable the device (refer to the case **NX_LINK_INITIALIZE** and **NX_LINK_ENABLE**). After the **NX_LINK_ENABLE** command is issued, the device should be ready to transmit and receive packets.

The IP instance transmits network packets via one of these commands:

Command	Description
NX__LINK__PACKET__SEND	An IPv4 or IPv6 packet is being transmitted,
NX__LINK__ARP__SEND	An ARP request or ARP response packet is being transmitted,
NX__LINK__ARP__RARP__SEND	A Reverse ARP request or response packet is being transmitted,

On processing these commands, the network driver needs to prepend the appropriate Ethernet frame header, and then send it to the underlying hardware for transmission. During the transmission process, the network driver has the exclusive ownership of the packet buffer area. Therefore once the data is being transmitted (or once the data has been copied into the driver internal transfer buffer), the network driver is responsible for releasing the packet buffer by first moving the prepend pointer past the Ethernet header to the IP header (and adjust packet length accordingly), and then by calling the ***nx__packet__transmit__release()*** service to release the packet. Not releasing the packet after data transmission will cause packets to leak.

The network device driver is also responsible for handling incoming data packets. In the RAM driver example, the received packet is processed by the function ***__nx__ram__network__driver__receive()***. Once the device receives an Ethernet frame, the driver is responsible for storing the data in NX__PACKET structure. Note that NetX Duo assumes the IP header starts from 4-byte aligned address. Since the length of Ethernet header is 14byte, the driver needs to store the starting of the Ethernet header at 2-byte aligned address to guarantee that the IP header starts at 4-byte aligned address.

TCP/IP Offload Driver Guidance

For TCP/IP offload feature, a driver function is needed for each IP interface. Here is a list of additional tasks for network driver.

- For command `NX_LINK_INITIALIZE`,
  - Create a driver thread to handle TCP/IP offload receive events.
- For command `NX_LINK_INTERFACE_ATTACH`,
  - Set the capability of to driver interface. See sample code below.

  ```
  driver_req_ptr -> nx_ip_driver_interface -> nx_interface_capability_flag = NX_INTERFACE
  ```
- For command `NX_LINK_ENABLE`,
  - Start the driver thread.
  - Set TCP/IP callback function to driver interface. See sample code below.

  ```
  driver_req_ptr -> nx_ip_driver_interface -> nx_interface_tcpip_offload_handler = _nx_dr
  ```

- For command `NX_LINK_DISABLE`,
    - Stop the driver thread
    - Clear TCP/IP callback function of driver interface.
- For command `NX_LINK_UNINITIALIZE`,
    - Delete the driver thread

TCP/IP Offload Driver Thread

The purpose of driver thread is to receive incoming TCP or UDP packets. In driver thread, there is typically a while loop to check whether there is incoming TCP or UDP packet available or connection established. When data are available, pass the TCP or UDP packet to NetX Duo. The room between `nx_packet_data_start` and `nx_packet_prepend_ptr` must be sufficient to insert TCP/IP header. For TCP socket, allocate packet with type `NX_TCP_PACKET`. For UDP socket, allocate packet with type `NX_UDP_PACKET`. Fill in incoming data from `nx_packet_append_ptr` to `nx_packet_data_end`. The data in `nx_packet_append_ptr` must contain TCP or UDP payload only. TCP/IP header **MUST** not be filled in packet. Adjust the packet length and set receive interface, then call `_nx_tcp_socket_driver_packet_receive` for TCP packet and `_nx_udp_socket_driver_packet_receive` for UDP packet. If a TCP connection is shutdown, call `_nx_tcp_socket_driver_packet_receive` with packet set to NULL. When connection is established, call `_nx_tcp_socket_driver_establish`.

TCP/IP Offload Driver Handler

The following driver commands are required for network interfaces with TCP/IP services. * For operation `NX_TCPIP_OFFLOAD_TCP_CLIENT_SOCKET_CONNECT`, * Allocate resource if needed. * Bind to local TCP port and connect to server. * Return success on connection established. When the connection is in progress, return `NX_IN_PROGRESS`. Or else, return failure. * For operation `NX_TCPIP_OFFLOAD_TCP_SERVER_SOCKET_LISTEN`, * Check for duplicate listen first. It can be called multiple time on same port. First time from `nx_tcp_server_socket_listen` and then `nx_tcp_server_socket_relisten`. * Allocate resource if needed. * Listen to local TCP port. * For operation `NX_TCPIP_OFFLOAD_TCP_SERVER_SOCKET_ACCEPT`, * Allocate resource if needed. * Accept connection. * For operation `NX_TCPIP_OFFLOAD_TCP_SERVER_SOCKET_UNLISTEN`, * Find TCP socket listening on local port. * Close the listening socket if found. * For operation `NX_TCPIP_OFFLOAD_TCP_SOCKET_DISCONNECT`, * Close the TCP/IP offload connection. * Unbind local TCP port. * Cleanup resources created during connect. * For operation `NX_TCPIP_OFFLOAD_TCP_SOCKET_SEND`, * Send data through TCP/IP offload. Be prepare to handle packet length larger than MSS or packet chain situation. * For operation `NX_TCPIP_OFFLOAD_UDP_SOCKET_BIND`, * Allocate resource if needed. * Bind to local UDP port. * For operation `NX_TCPIP_OFFLOAD_UDP_SOCKET_UNBIND`, * Unbind local UDP port. * Cleanup resources created during bind. * For operation `NX_TCPIP_OFFLOAD_UDP_SOCKET_SEND`, * Send data through TCP/IP offload. Be prepare to handle packet length larger than MTU or packet chain situation.

TSN driver support

TSN shapers are a series of hardware features that can be added to an Ethernet card equipped with TSN capabilities. We will discuss implementation of the PTP driver and the three types of shapers: CBS (Credit-Based Shaper), EST (Enhanced Scheduled Traffic), and FPE (Frame Preemption).

PTP initialize and callback function

PTP is utilized in various scenarios within the TSN system, particularly when the TAS shaper is enabled. Given the high requirements for the PTP clock, it is recommended to initialize PTP in Fine mode. Additionally, it is necessary to implement the PTP driver callbacks, which are invoked to get, set, and adjust the PTP clock.

the driver interface which is used to sychnorize ptp clock in the network.

```
UINT nx_driver_ptp_clock_callback(NX_PTP_CLIENT *client_ptr, UINT operation,
                                  NX_PTP_TIME *time_ptr, NX_PACKET *packet_ptr,
                                  VOID *callback_data)
```

operation	Description
NX_PTP_CLIENT_CLOCK_INIT	A driver PTP pointer is initialized,
NX_PTP_CLIENT_CLOCK_SET	Set PTP clock when syncorize PTP in the network
NX_PTP_CLIENT_CLOCK_PACKET_TS_EXTRACT	Get timestamp from packet,
NX_PTP_CLIENT_CLOCK_GET	Get timestamp from PTP clock,
NX_PTP_CLIENT_CLOCK_ADJUST	Adjust PTP clock by PTP offset when syncorize PTP in the network,
NX_PTP_CLIENT_CLOCK_PACKET_TS_PREPARE	Add NX_INTERFACE_CAPABILITY_PTP_TIMESTAMP support in interface,
NX_PTP_CLIENT_CLOCK_SOFT_TIMER_UPDATE	update soft timer, reserved for hardware driver,

Credit-based shaper (CBS) - IEEE 802.1Qav Forwarding and Queuing Enhancements for Time-Sensitive Stream

In general, a CBS works by assigning "credits" to each data packet. The number of credits a packet has determines when it can be transmitted. Packets with more credits are transmitted before packets with fewer credits. This allows the CBS to prioritize certain data streams over others, ensuring that high-priority data is transmitted first. The CBS driver needs to be implemented to support the CBS feature. The driver entry will be invoked from the application, passing application data to configure the CBS driver.

the driver entry:

```
UINT nx_driver_shaper_cbs_entry(NX_SHAPER_DRIVER_PARAMETER *parameter)
```

Data structure of driver entry parameter:

```
typedef struct NX_SHAPER_DRIVER_PARAMETER_STRUCT
{
    UINT          nx_shaper_driver_command;
    UCHAR         shaper_type;
    UCHAR         reserved[3];
    void         *shaper_parameter;
    NX_INTERFACE *nx_ip_driver_interface;
} NX_SHAPER_DRIVER_PARAMETER;
```

Data stucture of CBS parameter which define the necessary parameters that CBS driver needs.

```
typedef struct NX_SHAPER_CBS_PARAMETER_STRUCT
{
    INT   idle_slope;   /* Mbps */
    INT   send_slope;   /* Mbps */
    INT   hi_credit;
    INT   low_credit;
    UCHAR hw_queue_id;
    UCHAR reserved[3];
} NX_SHAPER_CBS_PARAMETER;
```

the driver entry parameter:

parameter -> nx_shaper_driver_command	Description
NX_SHAPER_COMMAND_INIT	initialization of enabling CBS,
NX_SHAPER_COMMAND_CONFIG	set hardware queue priority and capability of CBS,
NX_SHAPER_COMMAND_PARAMETER_SET	set parameter passed from application to driver,

Time-Aware Shaper (TAS) - IEEE 802.1Qbv Enhancements to Traffic Scheduling

A Time-Aware Shaper (TAS) is a mechanism used in Time-Sensitive Networking (TSN) systems to control the transmission of Ethernet frames based on the time. It's part of the IEEE 802.1Qbv standard. The TAS works by dividing time into repeating cycles, and each cycle is further divided into time intervals, or "gates". Each gate is either open or closed, and frames can only be transmitted when the gate is open. The TAS driver needs to be implemented to support the TAS feature. The driver entry will be invoked from the application, passing application data to configure the TAS driver.

the driver entry:

```
UINT nx_driver_shaper_tas_entry(NX_SHAPER_DRIVER_PARAMETER *parameter)
```

Data stucture of TAS parameter which define the necessary parameters that TAS driver needs.

```
typedef struct NX_SHAPER_TAS_PARAMETER_STRUCT
{
    ULONG64           base_time;
    UINT              cycle_time;
    UINT              cycle_time_extension;
    UINT              gcl_length;
    NX_SHAPER_TAS_GCL gcl[NX_SHAPER_GCL_LENGTH_MAX];
    void             *fp_parameter; /* Configured by shaper */
} NX_SHAPER_TAS_PARAMETER;
```

the driver entry parameter:

parameter -> nx__shaper__driver__command	Description
NX__SHAPER__COMMAND__INIT	initialization of enabling TAS,
NX__SHAPER__COMMAND__CONFIG	set hardware queue priority and capability of TAS,
NX__SHAPER__COMMAND__PARAMETER__SET	set parameter passed from application to driver,

Frame preemption (FPE) - 802.1Qbu

In traditional Ethernet networks, once a frame starts transmitting, it must be completely sent before another frame can begin. This can cause delays for time-sensitive data if it has to wait for a large, non-time-sensitive frame to finish transmitting. Frame Preemption addresses this issue by allowing a high-priority, time-sensitive frame to interrupt the transmission of a low-priority frame. The low-priority frame is then resumed after the high-priority frame has been sent. This ensures that time-sensitive data can be transmitted with minimal delay, even in a busy network The FPE driver needs to be implemented to support the FPE feature. The driver entry will be invoked from the application, passing application data to configure the FPE driver. the driver entry:

```
UINT nx_driver_shaper_fpe_entry(NX_SHAPER_DRIVER_PARAMETER *parameter)
```

Data structure of TAS parameter which define the necessary parameters that TAS driver needs.

```
typedef struct NX_SHAPER_FP_PARAMETER_STRUCT
{
    UCHAR verification_enable;          /* Enable/Disable fp verification (Application/Driv
    UCHAR express_queue_bitmap;         /* Bitmap of express queues */
    UCHAR express_guardband_enable;     /* Enable/Disable guard band on express queue */
    UCHAR reserved;
    UINT  ha;                           /* Hold advance time */
    UINT  ra;                           /* Release advance time */
} NX_SHAPER_FP_PARAMETER;
```

the driver entry parameter:

parameter -> nx__shaper__driver__command	Description
NX__SHAPER__COMMAND__INIT	initialization of enabling FPE,
NX__SHAPER__COMMAND__CONFIG	set hardware queue priority and capability of FPE,
NX__SHAPER__COMMAND__PARAMETER__SET	set parameter passed from application to driver,