NIC Device Driver

13TH APRIL, 2020

SAHIL SEMICONDUCTOR

Linux NIC Driver Introduction

NIC Driver

- Modern NICs communicate with the host system via PCI Express bus.
- PCI device implements Configuration Space that holds device specific information like vendor id, device id, type of mapping etc.
- Linux kernel uses these vendor id and device id to match the driver with the device.
- Packets are transmitted and received from the host to the NIC and vice versa through queues. Simpler devices like Intel e1000 use one queue for transmission and reception. Advanced devices like Mellanox Connect X5 use multiple queues.

Linux NIC Driver

- Two important structures are provided by the Linux kernel:

1. **struct net_device**
   - This is the structure by which any NIC device is represented in the kernel

2. **struct sk_buff**
   - This structure which wraps any packet that transits through a NIC

struct net_device

- This is the structure by which any NIC device is represented in the kernel
- The struct *net_device* belongs to the kernel data structures that needs to be allocated dynamically, having their own allocation function
- Network Interface is allocated by the kernel by calling *alloc_etherdev()*
- *After initializing members of net_device, register_netdev() is called to register the net_device with the kernel.*
- Unused network devices can be unregistered with *unregister_netdev()* and freed with *free_netdev()*, which also frees memory allocated for private data

net_device Operations

- Few net device operations that can be performed on the interface created by the *struct net_device:*

- **ndo_open:** This prepares and opens the interface. The interface is opened using *ifconfig <interface_name> up* cmd. In this method, the driver should request/map/register any system resource it needs (I/O ports, IRQ etc.), turn on the hardware and perform any other setup the device requires
- **ndo_stop:** The kernel executes this function when the interface is brought down using *ifconfig <interface_name>down*. This function should perform reverse operations of what has been done in *ndo_open()*
- **ndo_start_xmit:** The kernel executes this function when data needs to be transmitted through the device. This function takes the packet pointed to by skb and transmits it via the specified dev.
- **ndo_tx_timeout:** The kernel calls this method when a packet transmission fails to complete within a reasonable period, usually for dev->watchdog ticks. The driver should check what happened, handle the problem, and resume the packet transmission
- **net_device_stats:** This method returns the device statistic. This is what we see when netstat -i or ifconfig is executed

struct sk_buff

- This structure contains the headers for data that has been received or is about to be transmitted
- This structure is used by several different network layers.
- Various fields of the structure change as it is passes from one layer to another
- When socket buffer passed through the Application layer, it appends a header(FTP, HTTPS) before passing it to TCP layer
- TCP layer now appends its own header before passing it to IP layer, which obviously alters the data pointer in socket buffer
- Now the IP layer places it header in the socket buffer following the similar procedure used by the previous layers
- At this point, packet is received by driver which writes this in DMA buffer from which the device can read it.

Driver Initialization Flow

- Populate struct pci_device_id with the NIC device and vendor id.
- Declare struct pci_driver and register **probe** and **remove** callbacks. Also provide name of the driver and previously defined object of struct pci_device_id.
- Call pci_register_driver()
- When pci_register_driver() is called, kernel looks for the device which has the same vendor and device id as that provided by driver. If both ids match, probe function is invoked by the kernel (registered by the driver earlier).
- In probe function, the driver first determines the type of mapping the PCI device is offering (memory mapped IO or port IO).
- Then determine starting base address and length of mapping by functions pci_resource_start() and pci_resource_len() respectively.

Driver Initialization Flow (cont…)

- Request the kernel to exclusively grant read/write access to those memory regions to the driver.
- Once mapping is done, struct net_device should be allocated by calling alloc_etherdev()
- Initialize desired members (mtu, mac address etc.) of struct net_device
- Initialize NAPI by passing weight and NAPI polling function to netif_napi_add()
- Call register_netdev() to register the net_device with the kernel.
- Perform Device specific Initializations
- End of Probe function

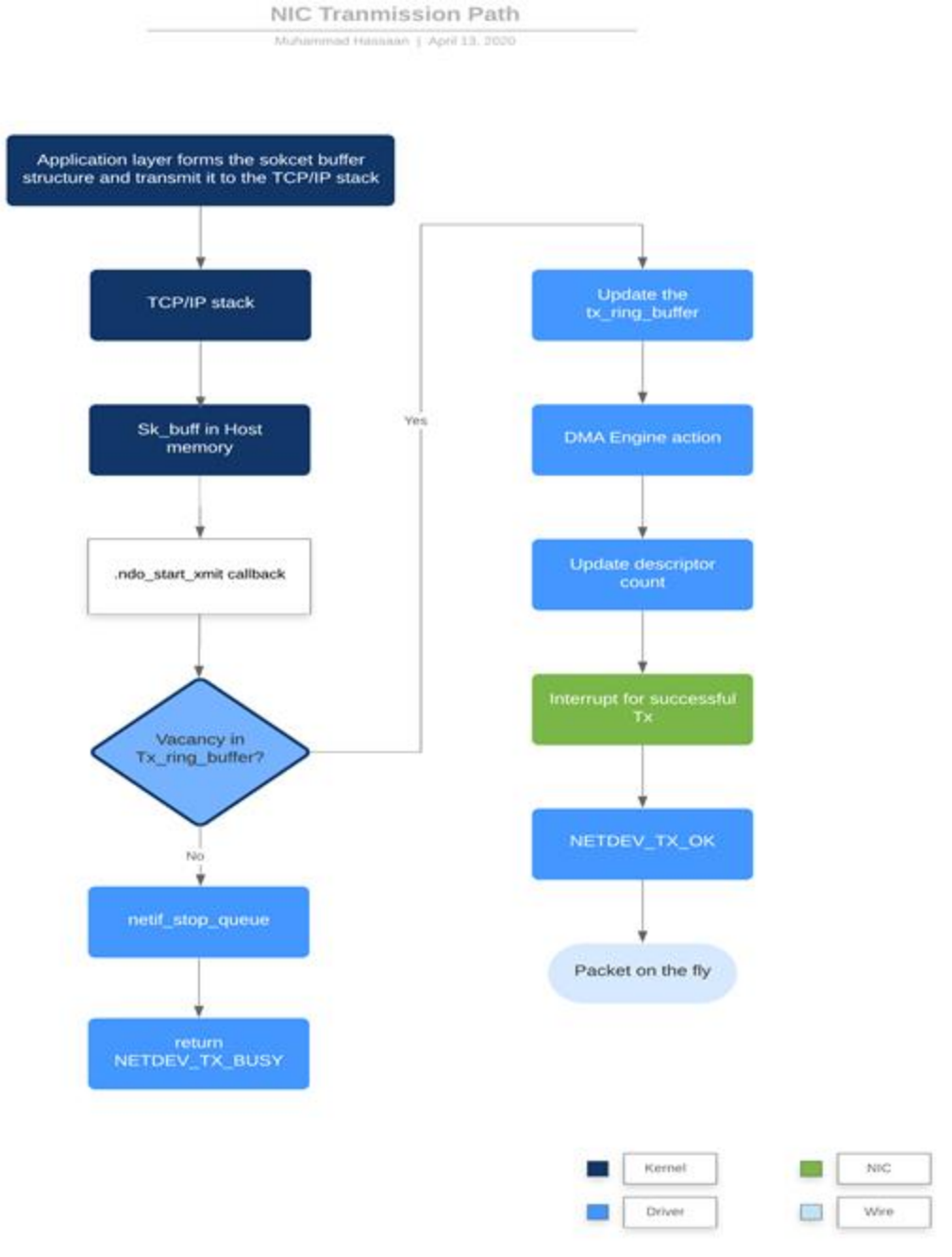
Driver Tx Path

- A socket is created which is the end point to receive/transmit data for the application
- The socket is bound to the port (like 8080)
- When data is passed from the application, it goes through the socket and is placed in the socket buffer allocated by the kernel.
- The socket buffer then passes through the TCP/IP stack where various headers are added (e.g. TCP header, IP header)
- As these headers are included, the data pointer in the socket buffers changes accordingly, making room for header, data and checksum data at the tail of the socket buffer.
- At this stage the .ndo_start_xmit callback from *net_device_ops* is called. This call back is registered by the driver as part of the initialization.

Driver Tx Path (cont…)

- *tx_ring_buffer* is checked through e1000_maybe_stop_tx() whether ring descriptors are available
- If the ring descriptor is found full; netif_stop_queue() is used to stop the flow of packets from the upper layers due to unavailability of the tx resources and NETDEV_TX_BUSY is returned
- On the availability of the ring descriptors the packet flow from the upper layer is started using netif_start_queue().
- Tx ring buffer is updated as the data is placed in the host memory
- The DMA engine then takes the data form host memory and place it into NIC memory
- After successful transmission, NIC generates interrupt to let the host know that packet has been copied from host to NIC.
- Host calls dev_kfree_skb_any() to free up the socket buffer and callback returns NETDEV_TX_OK

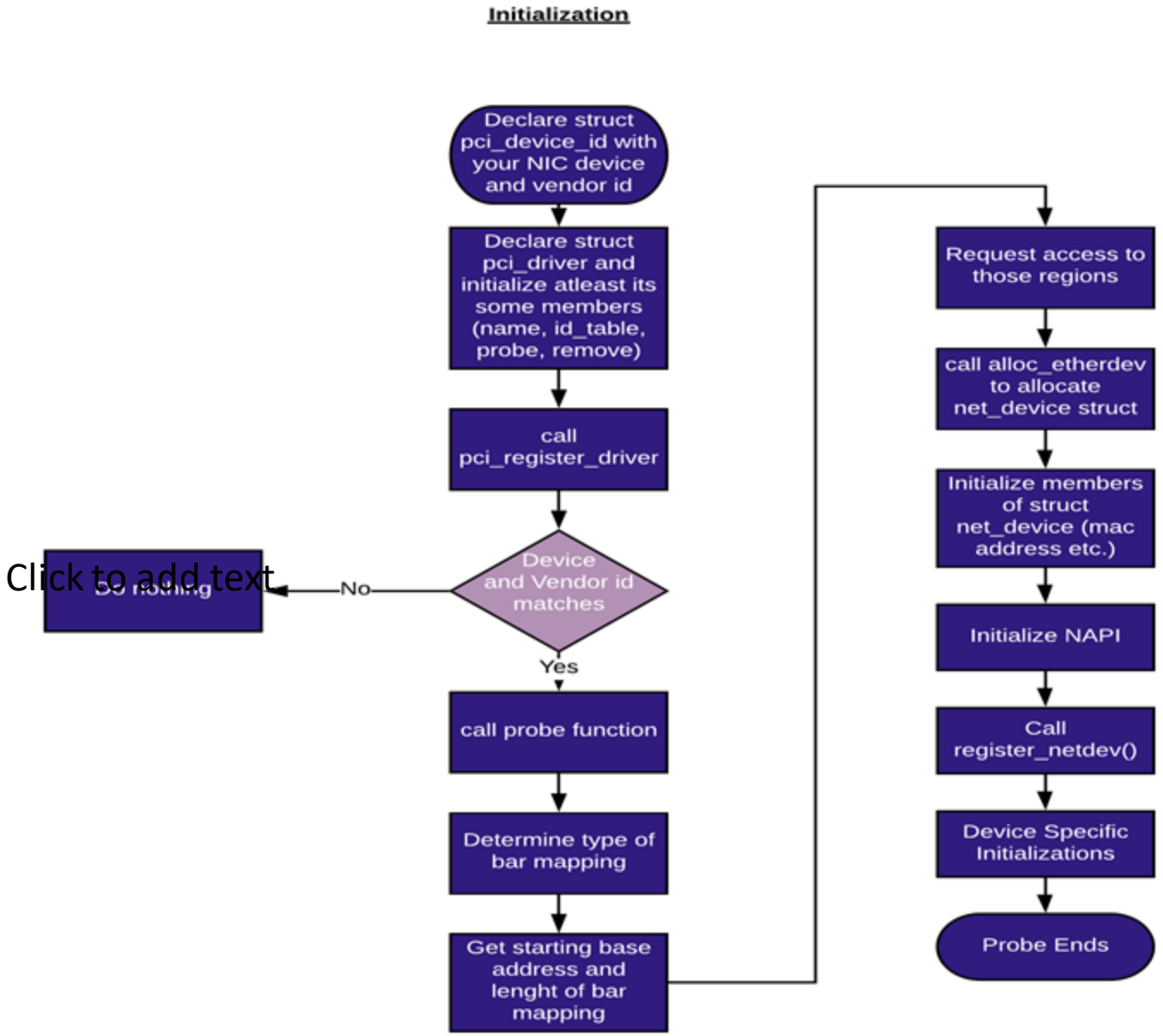
Driver Tx Path (cont…)



Driver Rx Path

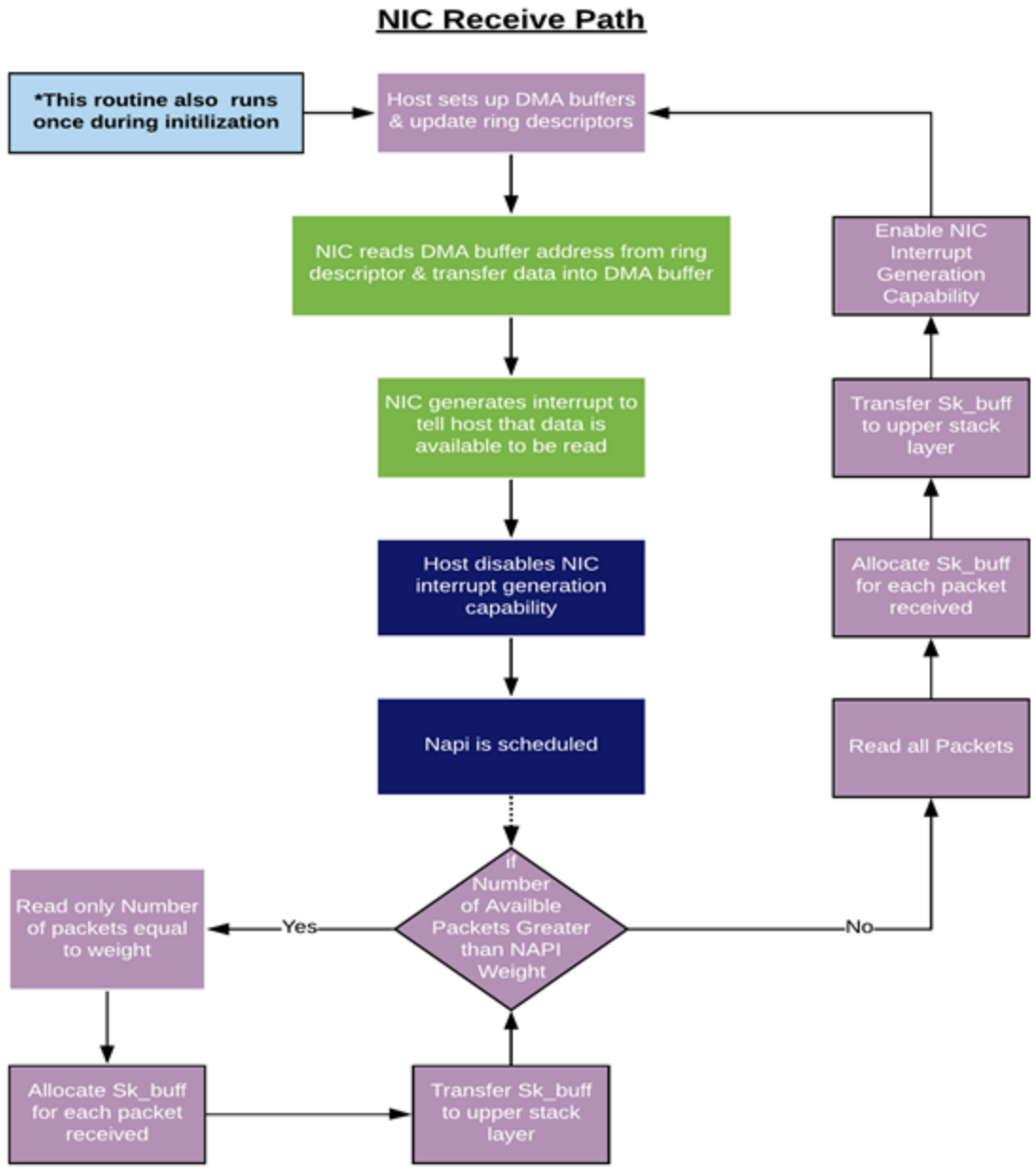
- Host sets up DMA Buffer and update receive ring descriptors
- The DMA engine transfers the data to the host memory and generates an interrupt
- The driver disables the interrupt and executes the interrupt handler which forms a socket buffer for the copied data and put the data into it
- Now, if the NAPI support is given in the driver; it is checked whether the number of packets to be read are within the defined NAPI weight
- If the no. of packets are greater than NAPI weight, the driver reads the defined no. of packets and push them to the upper networking stack and submits the remaining work to the kernel work queue.
- Upon the turn of the event as scheduled by the kernel, if the numbers of packets are less than NAPI weight all the packets are pushed to the host Network stack and the interrupts are enabled again otherwise, go to the previous stage

Driver Initialization Flow (cont…)



Click to add text

Driver Rx Path (cont…)



Ethtool Support

- Ethtool is a Network Interface Cards (NICs) utility/configuration tool. Ethtool allows you to query and change your NIC settings such as the Speed, Port, auto-negotiation etc.
- Ethtool function pointers are configured in net_device structure during the probe callback execution.
- Some of the common Ethtool functions are:

  - **get_drvinfo:** This function gives info about driver name, version, bus info and various supports(statistics, test, eeprom-access, register-dump, priv-flags) given by the driver
  - **get_link:** This function is called by the user app whenever the auto-negotiation restarts, if it is enabled
  - **get_ringparam:** This function gives the parameters of the Tx/Rx ring
  - **set_ringparam:** This function sets the parameter of the Tx/Rx ring
  - **set_phys_id:** This function sets the led on NIC hardware depending upon the state of the card to identify it.

Various NIC Offloads

- **Generic Segmentation offload (GSO)**
  - Segmentation of packet is done by Kernel in TCP layer
- **TCP Segmentation offload (TSO)**
  - Segmentation is done by NIC hardware
- **Generic Receive offload (GRO)**
  - Complement of GSO. Application layer reassembles fragmented packets
- **Large Receive offload (LRO)**
  - LRO is implemented in the driver to aggregate small packets into one large packet

Development Phases and High-Level Timelines

Development Phases

- Phase I (PF)
  - Basic Linux kernel NIC functionality which is capable of packet transmission and reception
  - Basic Ethtool support to check different configurations (since device currently doesn't support NIC functionality, most of this functionality will be handled by the driver without involving the device)

- Phase II (PF)
  - Extended functionality including
    - ✓ SRIOV Support
    - ✓ Different offloads

- Phase III (VF)
  - Linux kernel VF driver
    - ✓ This will allow running VMs with the device

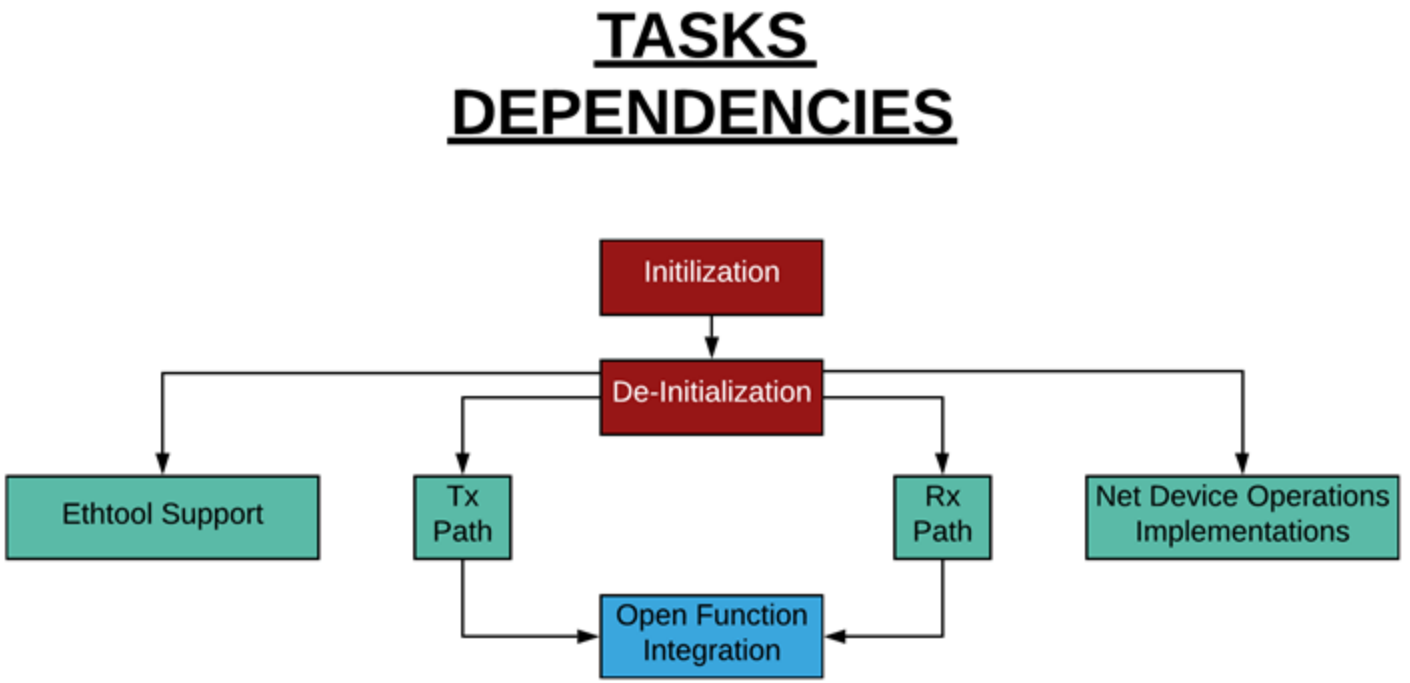
Phase I : Development Roadmap

- Six Major Tasks:
  - Driver and Device Initializations
  - Tx Path Implementation
  - Rx Path Implementation
  - Adding limited Ethtool support
  - Net device operations callbacks Implementation
  - De-initializations/Cleanup

Phase I : Assumption For Time Estimate

- Device is very similar to e1000 device.
- Device is verified and tested like loopback is verified to work without OVS.
- Code for entering OVS flow rules through CAM is working.
- Kernel version used is 5.3.0-45 (we may to add support for older kernels later)
- Dependency list may need to be revised.
- Buffer for troubleshooting/debugging is NOT ADDED due to lack of experience. (No time added for unit testing)
- Comments on Device specific Initialization and Rx, Tx path for QDMA IP will be given by Zafir after his study of Xilinx device driver
- Since the device doesn't support NIC functionality fully, certain configurations will be maintained within the drivers and will not be pushed to the device. This includes:
  - Reading/Setting interface MAC addresses
  - Link configuration (link speed, autoneg, FEC etc)
  - Interface stats

Phase I : Task Dependencies



The flowchart shows dependencies when coding and testing each block. In short, upper hierarchy have to be completed when starting to code and test lower blocks

Phase I : Initialization

- This requires Kernel specific Initializations (~2 man weeks)
- Various tasks include:
  - PCI and DMA Specific Initializations (BAR Mappings etc.)
  - Netdev struct initialization (MTU, speed, MAC address)
  - Interrupt handler registration and NAPI Initializations
  - Debugfs for Debugging
  - Net Device Open (Minimal Implementation)
  - Device Specific Initializations (Tentative time will by given by Zafir after Xilinx driver review)
  - Resetting device, enabling Tx and Rx etc.
  - SRIOV Initializations (Tentative time will be given once thorough understanding is developed)

Phase I : Tx Path

- Tx Data Path (~2 man weeks)
- Various tasks include:
  - Write socket buffer data coming from the upper stack layer into Host DMA Buffer
  - Update transmission ring descriptors so that NIC can read the data from Host DMA buffer (Implementation of open needed for Tx). Also update statistics

Phase I : Rx Path

- Rx Data Path (~2 man weeks)
- Various tasks include:
  - Interrupt Service Routine Implementation
  - Update Receive ring descriptors and allocate DMA buffers so that DMA engine can transfer data from NIC into Host memory. (Implementation of open needed for Rx)
  - Read Data from DMA buffer and transfer it to upper stack layer. Also update Statistics.

Phase I : Ethtool Support

- Basic Ethtool support (~3 man weeks)
- Various tasks include:
  - get_drvinfo
  - get_link
  - set_ringparam
  - get_ringparam
  - set_phys_id

Phase I : net_dev ops callbacks

- net device operation callbacks (~2.5 man weeks)
- Various tasks include:
  - Open (Integration from Initialization, Rx and Tx parts of open function)
  - start_xmit (Tx path callback)
  - change_mtu
  - set_mac_address
  - Stop (This is done in De-initialization)

Phase I : De-Initialization

- Unloading the driver (~2 man weeks)
- Various tasks include:
  - Give all the resources back to kernel
  - Remove function
  - Stop function

Phase II

- Some of the features to be supported in Phase II (requires device support):
  - Tx Checksum offload (L3/L4 checksum calculation)
  - Rx Checksum offload (L3/L4 checksum verification)
  - TSO Offload and UDP Segmentation offload
  - LRO offload
  - OVS control path/TC Flower offload
  - IPSEC offload
  - SRIOV Support
  - Receive flow steering
  - Device Health Checking
  - Self test (test loopback setup etc.)
  - Intelligent interrupt coalescence
  - Jumbo Frame Support
  - PHY Configurations
  - RDMA
  - SPI Flash Configuration
  - Overlay Network Support (Vxlan, NVGRE, GENEVE)

Phase III

- Some of the features to be supported in Phase III:
  - VF Linux kernel NIC driver
  - DPDK PF and VF drivers (current Xilinx DPDK PF and VF drivers may work with minimal changes)
  - VirtIO testing

- Some of Phase III features may be done earlier (once Phase I is complete).