Pensando Windows Design

# Overview

The Windows driver to support the Pensando NIC is based on a Windows NDIS Miniport driver. The NDIS wrapper on Windows allows for a more focused implementation on the requirements of a NIC port driver without having to implement the boiler plate features required within the Windows kernel. This document will cover the overall design and implementation of the Windows driver, including the NDIS interface features and areas which are common between Windows and Linux.

# DriverEntry

The DriverEntry routine is the topmost routine called by the Windows system when loading the driver. It is responsible for registering with the NDIS wrapper as well as any other global resource allocation and initialization. In the Pensando implementation, the driver performs NDIS wrapper registration as well as initialization of the control device interface for allowing user mode configuration tools to call into the driver.

## NDisMRegisterMiniportDriver()

This NDIS call is used by Windows network drivers to register a set of callbacks routines with the NDIS wrapper. This call also indicates the NDIS version which the driver will support (we are using 6.80 currently) with subsequent calls into the wrapper. The following sections cover the callbacks registered by the Pensando miniport driver as well as details into the implementation of the necessary support.

### SetOptionsHandler – SetOptions()

This callback is invoked to allow for miniport drivers to register any interfaces required outside of the primary NDIS interface. The Pensando driver supports the PnP interface and registers a set of callbacks to be invoked when a PnP state change occurs for the device. In particular, the IRP\_MN\_START\_DEVICE request is used by the driver to filter hardware resources assigned to the device.

See the section titled “PnP Support” for more information on this topic.

### UnloadHandler – DriverUnload()

This callback is invoked when the system is going to unload the driver. Except under some failure conditions, the Pensando driver is never unloaded during the boot cycle of the system.

### InitializeHandlerEx – InitializeEx()

This callback is the work horse of the NDIS interface and is responsible for initializing the underlying device, registering various attributes with the NDIS wrapper and processing hardware resources that are assigned to the device. Much of the implementation follows the processing found in the Linux routine ionic\_probe() with some additional Windows specific calls. The following description covers the general routines called during initialization processing.

1. SetRegistrationAttributes() – This is Windows specific processing to register attributes with the NDIS wrapper. This includes:
   1. Indicating the device is real hardware and not a virtual device
   2. The device supports bus master DMA operations
   3. The device is on the PCI bus
   4. Indicate the duration of the periodic timer the NDIS wrapper uses to determine if the miniport driver is hung.
2. ReadRegParameters() – This routine is used to read device specific registry information. In the case that a configuration parameter has changed, for example through the Advanced Options tab in the system configuration interface, NDIS will halt the device and then call the InitializeEx handler again. This allows the miniport driver to reread the configuration information to be used for the current setup.
3. Indicate to the NDIS wrapper the specific parameters for handling the scatter/gather processing. This includes:
   1. Registering a callback to be invoked when the system has allocated pages for the described request. This allows the miniport driver to insert these physical pages into the tx descriptors to perform the DMA operation. This callback is ionic\_process\_sg\_list().
   2. Indicate the maximum physical mapping allowed
   3. Indicate the device supports 64 bit addressing
4. Ionic\_map\_bars() – This call maps the BARs from the device into host memory.
5. Ionic\_register\_interrupts() – This routine registers the various interrupt callbacks for handling both MSI and legacy based interrupts. The callbacks which are registered with NDIS include:
   1. InterruptHandler – ionic\_isr\_legacy() – This is the ISR for handling legacy based interrupts from the device.
   2. InterruptDpcHandler – ionic\_dpc\_legacy() – This is the DPC handler for processing the DPC which is requested at the time the ISR fires.
   3. MessageInterruptHandler – ionic\_msi\_handler() – This is the MSI handler for processing MSIs from the device. A DPC is requested from this handler.
   4. MessageInterruptDpcHandler – ionic\_msi\_dpc\_handler() – This handles the DPC which was requested from the MSI handler. All the work performed for a given MSI Id is performed during this callback.
6. Ionic\_dev\_setup() – This routine maps out specific regions of each BAR which was previously mapped to host memory. It sets up pointers to the various registers as well as interrupt control regions in BAR0. It also reads from the register containing a signature for the firmware to ensure that it is valid.
7. Ionic\_identify() – This routine sends OS specific information to the device and receives device specific information from the device. The information it receives from the device include:
   1. ASIC revision
   2. Serial number
   3. Firmware version
   4. Number of logical interfaces and doorbells per LIf.
8. Ionic\_init() – This routine sends the CMD\_OPCODE\_INIT request to the device and waits for a successful completion.
9. Ionic\_port\_identify() – This routine sends the CMD\_OPCODE\_PORT\_IDENTIFY to the device and receives information about the physical port on the device. This information includes such things as speed and MTU size.
10. Ionic\_port\_init() – This routine sends the CMD\_OPCODE\_PORT\_INIT request to the device. It also allocates a region of shared memory for port configuration information and rx/tx stats for the port.
11. Ionic\_lif\_identify() – This routine retrieves information about the primary LIf including the MAC address, features and filter parameters.
12. Ionic\_lifs\_size() – This routine configures the number of tx and rx queues for the LIf. In the Windows implementation this is currently hard coded to setup 8 tx and 8 rx queues for handling requests. This routine will be modified to scale this number based on the number of available processors and the number of MSIs which have been allocated for the device.
13. Ionic\_lifs\_alloc() – This routine allocates the necessary memory resources for all the queues which will subsequently be initialized for the device. This allows the miniport driver to be certain most of the resources which will be required can be successfully allocated.
14. Ionic\_lifs\_init() – This routine initializes all the queues used for processing requests for the device. These include:
    1. AdminQ – This is used to process requests to the device in a more efficient manner than using the cmd processing used to this point.
    2. NotifQ – This queue is used to receive requests from the device such as link up/dn notifications.
    3. Rx/Tx queues – These are used to handle IO requests from/to the device.
15. SetGeneralAttribs() – This routine sets general parameters within the NDIS wrapper for the miniport driver. These attributes include:
    1. MTU size
    2. Maximum tx/rx speeds
    3. Mac options
    4. Supported filters
    5. Mac address – both the current and permanent addresses are provided.
16. Ionic\_open() – This routine enables the various rx/tx queues for general operation of the device.

### HaltHandlerEx – HaltEx()

This function is called by the NDIS wrapper to stop the miniport driver and free all allocated resources.

### PauseHandler – Pause()

This routine is called by the NDIS wrapper to pause all processing of requests by the miniport driver. All IO queues, OID requests or other active processing by the driver should be stopped before this routine returns to the wrapper.

### RestartHandler – Restart()

This routine is called by the NDIS wrapper to restart processing of active requests after a previous call to the PauseHandler has been invoked. The driver can restart IO queues as well as handle other requests.

### OidRequestHandler – OidRequest()

This routine is called by the NDIS wrapper to handle any set of the OID requests from other system or user mode components. The handler can be invoked for query, set and method type OID requests. Some of the operations requested by the NDIS wrapper are mandatory for miniport drivers.

### SendNetBufferListsHandler – ionic\_send\_packets()

This routine is invoked by the NDIS wrapper when a *net buffer list* is to be sent across the network by the device. This is the primary callback for transmission operations. When this routine is called by the NDIS wrapper to transmit a set of buffers, it is provided with a set of net buffer list control structures. These NBLs describe a set of net buffers, or NBs, for each fragment of the buffer to be transmitted.

The miniport driver performs some basic checks to ensure that tx is possible at the time of the callback and proceeds to walk the list of NBLs for the given request. For each NBL, it iterates through the list of NBs. For each of the NBs which it retrieves, the miniport driver will acquire an available txq packet from the preallocated ring of txq structures. At this point the miniport driver has reserved a txq packet which can be populated with physical pages from the NB for handling the DMA operation. To retrieve these pages for use in the DMA operations, the miniport driver calls the NdisAllocateNetBufferSGList() routine specifying the size of the NB to be handled.

Once the system has available resources to allocate these pages for the NB, it will invoke the callback registered during DMA configuration, ionic\_process\_sg\_list(). This routine will walk the list of s/g elements provided for the buffer and queue up each element to the txq descriptors. It does this by writing the physical page number of the page described by the s/g element into the txq descriptor and ringing the device doorbell for processing.

Note that currently the Windows driver handles at most 1 s/g element before copying each page of the described s/g list to a shared memory region of the device. This processing will be better optimized within the Windows driver in a future drop.

Once the entire NBL has been processed, or possibly within the completion of the processing of the entire NBL, the device will interrupt the host sending an MSI with Id of 1 to the miniport driver’s MSI handler callback. At this time, the driver will walk the txq and release the descriptors used for handling the request, returning them so they can be available for the next tx request.

### ReturnNetBufferListsHandler – ionic\_return\_packet()

This routine is called by the NDIS wrapper after the miniport driver has indicated a *net buffer list* after processing a receive request from the device. It is responsible for releasing any resources allocated during the receive processing.

For rx processing, the device will interrupt the host, calling the MSI handler with a Id of 1. When the MSI handler fires, and a DPC is queued for the request, the DPC handler will walk the current rxq descriptors. For each descriptor which requires processing, the miniport driver will initialize an NBL, described during tx processing, and once complete, will indicate these back to the NDIS wrapper for further processing.

Once the NDIS wrapper has completed processing the described NBL, it will call back into the ionic\_return\_packet() routine to indicate to the miniport driver that it can release any resources allocated for the provided NBL for the rx request.

### CancelSendHandler – CancelSend()

This routine is called by the NDIS wrapper to cancel a request initially sent by the SendNetBufferListsHandler callback. This is generally invoked for requests which have taken a longer time than normal to complete.

### DevicePnPEventNotifyHandler – DevicePnPEventNotify()

This callback is invoked during power change events on the system. It is always called immediately following the InitializeHandlerEx routine has returned to indicate a power state change to D3, full powered.

### ShutdownHandlerEx – ShutdownEx()

This routine is called when the system is shutting down. It allows the adapter to flush any packets to be sent, stop queue processing and generally allow the device to terminate operations prior to a full shutdown of the system.

### CancelOidRequestHandler – CancelOidRequest()

This routine is called to cancel outstanding Oid requests to the miniport driver. It is generally invoked for long pending requests that may be blocking forward progress of other components in the system.

## NdisRegisterDeviceEx()

This routine is called during DriverEntry processing by the Miniport driver to establish a control device, and symbolic link, to user mode. This allows for a user mode application to perform IOCtl calls into the driver for configuration. The current implementation uses this interface for configuring trace processing within the driver as well as retrieve the internal trace buffer from the driver.

# PnP Support

During the NDIS callback to set options within the miniport driver, the Pensando driver registers for PnP notifications. The miniport driver does this to allow for prefiltering of hardware resource requirements for the NIC. The callbacks which are registered for notification are described below.

## MiniportAddDeviceHandler – ionic\_add\_device()

This routine is called when the PnP manager adds a new device instance for the device described in the INF file to the system. The Pensando miniport driver allocates the adapter control structure for this device instance and initializes the information for further processing. This control structure is passed to all subsequent NDIS callbacks described above.

## MiniportRemoveDeviceHandler – ionic\_remove\_device()

This routine is called when the device is removed from the system. Since the Pensando NIC does not support PnP capabilities, this routine is not called during normal operation.

## MiniportFilterResourceRequirementsHandler – ionic\_filter\_resource\_requirements()

This routine is invoked to allow for the miniport driver to filter any hardware resources assigned to the driver. In the Pensando driver this routine assigns a processor affinity to each MSI Id assigned to the driver. When the NDIS driver later calls the InitializeEx handler and passes in the assigned resources, the correct affinity is set for each MSI Id.

## MiniportStartDeviceHandler – ionic\_start\_device()

This callback is invoked by the NDIS wrapper when an IRP\_MN\_START\_DEVICE request is sent by the PnP manager. The Pensando miniport driver does not perform any actions during this callback.

# Installation Process

1. Copy ionic64.sys, Ionic64.cat and Ionic64.inf to the test system. Note that a DEBUG build of the driver should be used during this phase of development.
   1. As discussed on the call, if you are doing a build of the driver you will need to either install the corresponding test cert on the test system, or, perform an Attestation Portal signing using your EV cert.
2. Open Device Manager and find the Pensando Ethernet device which requires installation. DO NOT install the driver for the management device, only the ethernet device with PID 1002.
3. Right click the device and select "Update Driver".
4. Select "Browse my computer ...".
5. Select "Let me pick from a list ...".
6. Select "Next".
7. Select "Have Disk".
8. Browse to location of above files copied to test system and select the INF file.
9. Select 'OK'
10. Select "Next".
11. Select "Install this driver software anyway".
12. Select "Close".

# At this point the driver software is installed. For the current implementation, reboot the system at this point. When the system reboots, go to Device Manager. Under Network Adapters you should see the 2 Pensando entries for the NIC. At this point the connections can be configured with an IP address and tested with ping or mapping a network share over CIFS.

# Ionic Trace Subsystem

The trace subsystem for the Ionic driver allows for quick configuration and gathering of information about the function of the driver. The tool which is used to configure and gather this trace information is currently a command line tool called IonicConfig.exe but can be easily built into any user mode component. The basic attributes of the trace subsystem are:

* 4 different levels of trace are offered that include:
  + ERROR
  + WARNING
  + VERBOSE
  + VERBOSE\_2
* An extensible set of categories within the driver that can be filtered. These are currently listed in the UserCommon.h header file. The categories can be combined so that multiple subsystems can be traced within the driver.
* An in-memory buffer is used to contain the trace information. The buffer is configurable and because it is allocated in kernel memory, is contained within a kernel memory crash dump for post mortem analysis.
* Using the IonicConfig tool, the trace buffer can be retrieved during normal execution of the system.
* The overhead of the trace subsystem, except for the VERBOSE and VERBOSE\_2 levels, has almost no impact on performance. These latter 2 levels can incur overhead due to their extensive tracing in performance critical paths such as scatter/gather processing.

To use the trace subsystem, run the IonicConfig tool from an Administrator command prompt. Running with no parameters will show a list of options for the tool. These options are:

* Level: This specifies the level of trace to perform and are described above
* Component: These is a value indicating which components, or categories, within the driver to trace. This value can be a hexadecimal value combining multiple components are listed in the UserCommon.h header file
* Length: This is the length of the internal trace buffer in KB. The maximum value is currently 10 \* 1024, or 10MB after conversion from KB.
* GetTrace: This parameter will override any other parameter set and will retrieve the current content of the trace buffer.
* Flags: This parameter is not indicated in the help listing but if set to a value of 1, will dump out all current trace copied into the trace buffer to the WinDbg output as well.

Note that the trace configuration is stored in the registry to facilitate the capture of trace during driver load and initialization time. Setting the necessary trace parameters and rebooting the system will allow capture of this early processing normally not accessible without the user of a kernel debugger such as WinDbg.