Linux kernel driver for Elastic Network Adapter (ENA) family

Overview

ENA is a networking interface designed to make good use of modern CPU features and system architectures.

The ENA device exposes a lightweight management interface with a minimal set of memory mapped registers and extendible command set through an Admin Queue.

The driver supports a range of ENA devices, is link-speed independent (i.e., the same driver is used for 10GbE, 25GbE, 40GbE, etc), and has a negotiated and extendible feature set.

Some ENA devices support SR-IOV. This driver is used for both the SR-IOV Physical Function (PF) and Virtual Function (VF) devices.

ENA devices enable high speed and low overhead network traffic processing by providing multiple Tx/Rx queue pairs (the maximum number is advertised by the device via the Admin Queue), a dedicated MSI-X interrupt vector per Tx/Rx queue pair, adaptive interrupt moderation, and CPU cacheline optimized data placement.

The ENA driver supports industry standard TCP/IP offload features such as checksum offload. Receive-side scaling (RSS) is supported for multi-core scaling.

The ENA driver and its corresponding devices implement health monitoring mechanisms such as watchdog, enabling the device and driver to recover in a manner transparent to the application, as well as debug logs.

Some of the ENA devices support a working mode called Low-latency Queue (LLQ), which saves several more microseconds.
ENA Source Code Directory Structure ═══════════════════════════════════

ena_com.[ch]	Management communication layer. This layer is responsible for the handling all the management (admin) communication between the device and the driver.
ena_eth_com.[ch]	Tx/Rx data path.
ena_admin_defs.h	Definition of ENA management interface.
ena_eth_io_defs.h	Definition of ENA data path interface.
ena_common_defs.h	Common definitions for ena_com layer.
ena_regs_defs.h	Definition of ENA PCI memory-mapped (MMIO) registers.
ena_netdev.[ch]	Main Linux kernel driver.
ena_ethtool.c	ethtool callbacks.
ena_pci_id_tbl.h	Supported device IDs.

Management Interface:

ENA management interface is exposed by means of:

- PCIe Configuration Space
- Device Registers
- Admin Queue (AQ) and Admin Completion Queue (ACQ)
- Asynchronous Event Notification Queue (AENQ)

ENA device MMIO Registers are accessed only during driver initialization and are not used during further normal device operation.

AQ is used for submitting management commands, and the results/responses are reported asynchronously through ACQ.

ENA introduces a small set of management commands with room for vendor-specific extensions. Most of the management operations are framed in a generic Get/Set feature command.

The following admin queue commands are supported:

- Create I/O submission queue
- Create I/O completion queue
- Destroy I/O submission queue
- Destroy I/O completion queue
- Get feature
- Set feature
- Configure AENQ
- Get statistics

Refer to ena_admin_defs.h for the list of supported Get/Set Feature properties.

The Asynchronous Event Notification Queue (AENQ) is a uni-directional queue used by the ENA device to send to the driver events that cannot be reported using ACQ. AENQ events are subdivided into groups. Each group may have multiple syndromes, as shown

below

The events are:

Group	Syndrome
Link state change	**X**
Fatal error	**X**
Notification	Suspend traffic
Notification	Resume traffic
Keep-Alive	**X**

ACQ and AENQ share the same MSI-X vector.

Keep-Alive is a special mechanism that allows monitoring the device's health. A Keep-Alive event is delivered by the device every second. The driver maintains a watchdog (WD) handler which logs the current state and statistics. If the keep-alive events aren't delivered as expected the WD resets the device and the driver.

Data Path Interface

I/O operations are based on Tx and Rx Submission Queues (Tx SQ and Rx SQ correspondingly). Each SQ has a completion queue (CQ) associated with it.

The SQs and CQs are implemented as descriptor rings in contiguous physical memory.

The ENA driver supports two Queue Operation modes for Tx SQs:

- **Regular mode:** In this mode the Tx SQs reside in the host's memory. The ENA device fetches the ENA Tx descriptors and packet data from host memory.
- **Low Latency Queue (LLQ) mode or "push-mode":** In this mode the driver pushes the transmit descriptors and the first 96 bytes of the packet directly to the ENA device memory space. The rest of the packet payload is fetched by the device. For this operation mode, the driver uses a dedicated PCI device memory BAR, which is mapped with write-combine capability.

  Note that not all ENA devices support LLQ, and this feature is negotiated with the device upon initialization. If the ENA device does not support LLQ mode, the driver falls back to the regular mode.

The Rx SQs support only the regular mode.

The driver supports multi-queue for both Tx and Rx. This has various benefits:

- Reduced CPU/thread/process contention on a given Ethernet interface.
- Cache miss rate on completion is reduced, particularly for data cache lines that hold the sk_buff structures.
- Increased process-level parallelism when handling received packets.
- Increased data cache hit rate, by steering kernel processing of packets to the CPU, where the application thread consuming the packet is running.
- In hardware interrupt re-direction.

Interrupt Modes

The driver assigns a single MSI-X vector per queue pair (for both Tx and Rx directions). The driver assigns an additional dedicated MSI-X vector for management (for ACQ and AENQ).

Management interrupt registration is performed when the Linux kernel probes the adapter, and it is de-registered when the adapter is removed. I/O queue interrupt registration is performed when the Linux interface of the adapter is opened, and it is de-registered when the interface is closed.

The management interrupt is named:

```
ena-mgmnt@pci:<PCI domain:bus:slot.function>
```

and for each queue pair, an interrupt is named:

```
<interface name>-Tx-Rx-<queue index>
```

The ENA device operates in auto-mask and auto-clear interrupt modes. That is, once MSI-X is delivered to the host, its Cause bit is automatically cleared and the interrupt is masked. The interrupt is unmasked by the driver after NAPI processing is complete.

Interrupt Moderation

ENA driver and device can operate in conventional or adaptive interrupt moderation mode.

In conventional mode the driver instructs device to postpone interrupt posting according to static interrupt delay value. The interrupt delay value can be configured through *ethtool(8)*. The following *ethtool* parameters are supported by the driver: `tx-usecs`, `rx-usecs`

In adaptive interrupt moderation mode the interrupt delay value is updated by the driver dynamically and adjusted every NAPI cycle according to the traffic nature.

Adaptive coalescing can be switched on/off through *ethtool(8)*'s `adaptive_rx on|off` parameter.

More information about Adaptive Interrupt Moderation (DIM) can be found in Documentation/networking/net_dim.rst

RX copybreak

The rx_copybreak is initialized by default to ENA_DEFAULT_RX_COPYBREAK and can be configured by the ETHTOOL_STUNABLE command of the SIOCETHTOOL ioctl.

Statistics

The user can obtain ENA device and driver statistics using *ethtool*. The driver can collect regular or extended statistics (including per-queue stats) from the device.

In addition the driver logs the stats to syslog upon device reset.

MTU

The driver supports an arbitrarily large MTU with a maximum that is negotiated with the device. The driver configures MTU using the SetFeature command (ENA_ADMIN_MTU property). The user can change MTU via *ip(8)* and similar legacy tools.

Stateless Offloads

The ENA driver supports:

- IPv4 header checksum offload
- TCP/UDP over IPv4/IPv6 checksum offloads

RSS

- The ENA device supports RSS that allows flexible Rx traffic steering.
- Toeplitz and CRC32 hash functions are supported.
- Different combinations of L2/L3/L4 fields can be configured as inputs for hash functions.
- The driver configures RSS settings using the AQ SetFeature command (ENA_ADMIN_RSS_HASH_FUNCTION, ENA_ADMIN_RSS_HASH_INPUT and ENA_ADMIN_RSS_INDIRECTION_TABLE_CONFIG properties).
- If the NETIF_F_RXHASH flag is set, the 32-bit result of the hash function delivered in the Rx CQ descriptor is set in the received SKB.
- The user can provide a hash key, hash function, and configure the indirection table through *ethtool(8)*.

DATA PATH

Tx

`ena_start_xmit()` is called by the stack. This function does the following:

- Maps data buffers (`skb->data` and frags).
- Populates `ena_buf` for the push buffer (if the driver and device are in push mode).
- Prepares ENA bufs for the remaining frags.
- Allocates a new request ID from the empty `req_id` ring. The request ID is the index of the packet in the Tx info. This is used for out-of-order Tx completions.
- Adds the packet to the proper place in the Tx ring.
- Calls `ena_com_prepare_tx()`, an ENA communication layer that converts the `ena_bufs` to ENA descriptors (and adds meta ENA descriptors as needed).
    - This function also copies the ENA descriptors and the push buffer to the Device memory space (if in push mode).
- Writes a doorbell to the ENA device.
- When the ENA device finishes sending the packet, a completion interrupt is raised.
- The interrupt handler schedules NAPI.
- The `ena_clean_tx_irq()` function is called. This function handles the completion descriptors generated by the ENA, with a single completion descriptor per completed packet.
    - `req_id` is retrieved from the completion descriptor. The `tx_info` of the packet is retrieved via the `req_id`. The data buffers are unmapped and `req_id` is returned to the empty `req_id` ring.
    - The function stops when the completion descriptors are completed or the budget is reached.

Rx

- When a packet is received from the ENA device.
- The interrupt handler schedules NAPI.
- The `ena_clean_rx_irq()` function is called. This function calls `ena_com_rx_pkt()`, an ENA communication layer function, which returns the number of descriptors used for a new packet, and zero if no new packet is found.
- `ena_rx_skb()` checks packet length:
    - If the packet is small (len < rx_copybreak), the driver allocates a SKB for the new packet, and copies the packet payload into the SKB data buffer.
        - In this way the original data buffer is not passed to the stack and is reused for future Rx packets.
    - Otherwise the function unmaps the Rx buffer, sets the first descriptor as *skb*'s linear part and the other descriptors as the *skb*'s frags.
- The new SKB is updated with the necessary information (protocol, checksum hw verify result, etc), and then passed to the network stack, using the NAPI interface function `napi_gro_receive()`.