APPENDIX A

Linux API

In this appendix I cover the two most fundamental data structures in the Linux Kernel Networking stack: the `sk_buff` and the `net_device`. This is reference material that can help when reading the rest of this book, as you will probably encounter these two structures in almost every chapter. Becoming familiar with and learning about these two data structures is essential for understanding the Linux Kernel Networking stack. Subsequently, there is a section about remote DMA (RDMA), which is further reference material for Chapter 13. It describes in detail the main methods and the main data structures that are used by RDMA. This appendix is a good place to always return to, especially when looking for definitions of the basic terms.

The sk_buff Structure

The `sk_buff` structure represents a packet. SKB stands for *socket buffer*. A packet can be generated by a local socket in the local machine, which was created by a userspace application; the packet can be sent outside or to another socket in the same machine. A packet can also be created by a kernel socket; and you can receive a physical frame from a network device (Layer 2) and attach it to an `sk_buff` and pass it on to Layer 3. When the packet destination is your local machine, it will continue to Layer 4. If the packet is not for your machine, it will be forwarded according to your routing tables rules, if your machine supports forwarding. If the packet is damaged for any reason, it will be dropped. The `sk_buff` is a very large structure; I mention most of its members in this section. The `sk_buff` structure is defined in `include/linux/skbuff.h`. Here is a description of most of its members:

- `ktime_t tstamp`

  Timestamp of the arrival of the packet. Timestamps are stored in the SKB as offsets to a base timestamp. Note: do not confuse `tstamp` of the SKB with hardware timestamping, which is implemented with the `hwtstamps` of `skb_shared_info`. I describe the `skb_shared_info` object later in this appenidx.

  Helper methods:

  - `skb_get_ktime(const struct sk_buff *skb)`: Returns the `tstamp` of the specified `skb`.
  - `skb_get_timestamp(const struct sk_buff *skb, struct timeval *stamp)`: Converts the offset back to a `struct timeval`.
  - `net_timestamp_set(struct sk_buff *skb)`: Sets the timestamp for the specified `skb`. The timestamp calculation is done with the `ktime_get_real()` method, which returns the time in `ktime_t` format.
  - `net_enable_timestamp()`: This method should be called to enable SKB timestamping.
  - `net_disable_timestamp()`: This method should be called to disable SKB timestamping.

- `struct sock *sk`

The socket that owns the SKB, for local generated traffic and for traffic that is destined for the local host. For packets that are being forwarded, `sk` is NULL. Usually when talking about sockets you deal with sockets which are created by calling the `socket()` system call from userspace. It should be mentioned that there are also kernel sockets, which are created by calling the `sock_create_kern()` method. See for example in `vxlan_init_net()` in the VXLAN driver, `drivers/net/vxlan.c`.

Helper method:

- `skb_orphan(struct sk_buff *skb)`: If the specified `skb` has a destructor, call this destructor; set the sock object (`sk`) of the specified `skb` to NULL, and set the destructor of the specified `skb` to NULL.

- `struct net_device *dev`

  The `dev` member is a `net_device` object which represents the network interface device associated to the SKB; you will sometimes encounter the term NIC (Network Interface Card) for such a network device. It can be the network device on which the packet arrives, or the network device on which the packet will be sent. The `net_device` structure will be discussed in depth in the next section.

- `char cb[48]`

  This is the control buffer. It is free to use by any layer. This is an opaque area used to store private information. For example, the TCP protocol uses it for the TCP control buffer:

```
#define TCP_SKB_CB(__skb) ((struct tcp_skb_cb *)&((__skb)->cb[0]))
(include/net/tcp.h)
```

  The Bluetooth protocol also uses the control block:

```
#define bt_cb(skb) ((struct bt_skb_cb *)((skb)->cb))
    (include/net/bluetooth/bluetooth.h)
```

- `unsigned long _skb_refdst`

  The destination entry (`dst_entry`) address. The `dst_entry struct` represents the routing entry for a given destination. For each packet, incoming or outgoing, you perform a lookup in the routing tables. Sometimes this lookup is called FIB lookup. The result of this lookup determines how you should handle this packet; for example, whether it should be forwarded, and if so, on which interface it should be transmitted; or should it be thrown, should an ICMP error message be sent, and so on. The `dst_entry` object has a reference counter (the `__refcnt` field). There are cases when you use this reference count, and there are cases when you do not use it. The `dst_entry` object and the lookup in the FIB is discussed in more detail in Chapter 4.

  Helper methods:

  - `skb_dst_set(struct sk_buff *skb, struct dst_entry *dst)`: Sets the `skb dst`, assuming a reference was taken on `dst` and should be released by the `dst_release()` method (which is invoked by the `skb_dst_drop()` method).

- skb_dst_set_noref(struct sk_buff *skb, struct dst_entry *dst): Sets the skb dst, assuming a reference was not taken on dst. In this case, the skb_dst_drop() method will not call the dst_release() method for the dst.

■ **Note** The SKB might have a dst_entry pointer attached to it; it can be reference counted or not. The low order bit of _skb_refdst is set if the reference counter was not taken.

- struct sec_path *sp

  The security path pointer. It includes an array of IPsec XFRM transformations states (xfrm_state objects). IPsec (IP Security) is a Layer 3 protocol which is used mostly in VPNs. It is mandatory in IPv6 and optional in IPv4. Linux, like many other operating systems, implements IPsec both for IPv4 and IPv6. The sec_path structure is defined in include/net/xfrm.h. See more in Chapter 10, which deals with the IPsec subsystem.

  Helper method:

  - struct sec_path *skb_sec_path(struct sk_buff *skb): Returns the sec_path object (sp) associated with the specified skb.

- unsigned int len

  The total number of packet bytes.

- unsigned int data_len

  The data length. This field is used only when the packet has nonlinear data (paged data).

  Helper method:

  - skb_is_nonlinear(const struct sk_buff *skb): Returns true when the data_len of the specified skb is larger than 0.

- __u16 mac_len

  The length of the MAC (Layer 2) header.

- __wsum csum

  The checksum.

- __u32 priority

  The queuing priority of the packet. In the Tx path, the priority of the SKB is set according to the socket priority (the sk_priority field of the socket). The socket priority in turn can be set by calling the setsockopt() system call with the SO_PRIORITY socket option. Using the net_prio cgroup kernel module, you can define a rule which will set the priority for the SKB; see in the description of the sk_buff netprio_map field, later in this section, and also in Documentation/cgroup/netprio.txt. For forwarded packets, the priority is set according to TOS (Type Of Service) field in the IP header. There is a table named ip_tos2prio which consists of 16 elements. The mapping from TOS to priority is done by the rt_tos2priority() method, according to the TOS field of the IP header; see the ip_forward() method in net/ipv4/ip_forward.c and the ip_tos2prio definition in include/net/route.h.

- `__u8 local_df:1`

  Allow local fragmentation flag. If the value of the `pmtudisc` field of the socket which sends the packet is IP_PMTUDISC_DONT or IP_PMTUDISC_WANT, `local_df` is set to 1; if the value of the `pmtudisc` field of the socket is IP_PMTUDISC_DO or IP_PMTUDISC_PROBE, `local_df` is set to 0. See the implementation of the `__ip_make_skb()` method in `net/ipv4/ip_output.c`. Only when the packet `local_df` is 0 do you set the IP header don't fragment flag, IP_DF; see the `ip_queue_xmit()` method in `net/ipv4/ip_output.c`:

  ```
   . . .
    if (ip_dont_fragment(sk, &rt->dst) && !skb->local_df)
           iph->frag_off = htons(IP_DF);
       else
         iph->frag_off = 0;
   . . .
  ```

  The `frag_off` field in the IP header is a 16-bit field, which represents the offset and the flags of the fragment. The 13 leftmost (MSB) bits are the offset (the offset unit is 8-bytes) and the 3 rightmost (LSB) bits are the flags. The flags can be IP_MF (there are more fragments), IP_DF (do not fragment), IP_CE (for congestion), or IP_OFFSET (offset part).

  The reason behind this is that there are cases when you do not want to allow IP fragmentation. For example, in Path MTU Discovery (PMTUD), you set the DF (don't fragment) flag of the IP header. Thus, you don't fragment the outgoing packets. Any network device along the path whose MTU is smaller than the packet will drop it and send back an ICMP packet ("Fragmentation Needed"). Getting these ICMP "Fragmentation Needed" packets is required in order to determine the Path MTU. See more in Chapter 3. From userspace, setting IP_PMTUDISC_DO is done, for example, thus (the following code snippet is taken from the source code of the `tracepath` utility from the `iputils` package; the `tracepath` utility finds the path MTU):

  ```
  . . .
  int on = IP_PMTUDISC_DO;
  setsockopt(fd, SOL_IP, IP_MTU_DISCOVER, &on, sizeof(on));
  . . .
  ```

- `__u8 cloned:1`

  When the packet is cloned with the `__skb_clone()` method, this field is set to 1 in both the cloned packet and the primary packet. Cloning SKB means creating a private copy of the `sk_buff struct`; the data block is shared between the clone and the primary SKB.

- `__u8 ip_summed:2`

  Indicator of IP (Layer 3) checksum; can be one of these values:

  - CHECKSUM_NONE: When the device driver does not support hardware checksumming, it sets the `ip_summed` field to be CHECKSUM_NONE. This is an indication that checksumming should be done in software.
  - CHECKSUM_UNNECESSARY: No need for any checksumming.

  - CHECKSUM_COMPLETE: Calculation of the checksum was completed by the hardware, for incoming packets.
  - CHECKSUM_PARTIAL: A partial checksum was computed for outgoing packets; the hardware should complete the checksum calculation. CHECKSUM_COMPLETE and CHECKSUM_PARTIAL replace the CHECKSUM_HW flag, which is now deprecated.

- `__u8 nohdr:1`

  Payload reference only, must not modify header. There are cases when the owner of the SKB no longer needs to access the header at all. In such cases, you can call the `skb_header_release()` method, which sets the `nohdr` field of the SKB; this indicates that the header of this SKB should not be modified.

- `__u8 nfctinfo:3`

  Connection Tracking info. Connection Tracking allows the kernel to keep track of all logical network connections or sessions. NAT relies on Connection Tracking information for its translations. The value of the `nfctinfo` field corresponds to the `ip_conntrack_info` `enum` values. So, for example, when a new connection is starting to be tracked, the value of `nfctinfo` is IP_CT_NEW. When the connection is established, the value of `nfctinfo` is IP_CT_ESTABLISHED. The value of `nfctinfo` can change to IP_CT_RELATED when the packet is related to an existing connection—for example, when the traffic is part of some FTP session or SIP session, and so on. For a full list of `ip_conntrack_info enum` values see `include/uapi/linux/netfilter/nf_conntrack_common.h`. The `nfctinfo` field of the SKB is set in the `resolve_normal_ct()` method, `net/netfilter/nf_conntrack_core.c`. This method performs a Connection Tracking lookup, and if there is a miss, it creates a new Connection Tracking entry. Connection Tracking is discussed in depth in Chapter 9, which deals with the netfilter subsystem.

- `__u8 pkt_type:3`

  For Ethernet, the packet type depends on the destination MAC address in the ethernet header, and is determined by the `eth_type_trans()` method:

  - PACKET_BROADCAST for broadcast
  - PACKET_MULTICAST for multicast
  - PACKET_HOST if the destination MAC address is the MAC address of the device which was passed as a parameter
  - PACKET_OTHERHOST if these conditions are not met

  See the definition of the packet types in `include/uapi/linux/if_packet.h`.

- `__u8 ipvs_property:1`

  This flag indicates whether the SKB is owned by `ipvs` (IP Virtual Server), which is a kernel-based transport layer load-balancing solution. This field is set to 1 in the transmit methods of `ipvs` (`net/netfilter/ipvs/ip_vs_xmit.c`).

- `__u8 peeked:1`

  This packet has been already seen, so stats have been done for it—so don't do them again.

- `__u8 nf_trace:1`

  The netfilter packet trace flag. This flag is set by the packet flow tracing the netfilter module, xt_TRACE module, which is used to mark packets for tracing (`net/netfilter/xt_TRACE.c`).

  Helper method:

  - `nf_reset_trace(struct sk_buff *skb)`: Sets the `nf_trace` of the specified `skb` to 0.

- `__be16 protocol`

  The protocol field is initialized in the Rx path by the `eth_type_trans()` method to be ETH_P_IP when working with Ethernet and IP.

- `void (*destructor)(struct sk_buff *skb)`

  A callback that is invoked when freeing the SKB by calling the `kfree_skb()` method.

- `struct nf_conntrack *nfct`

  The associated Connection Tracking object, if it exists. The `nfct` field, like the `nfctinfo` field, is set in the `resolve_normal_ct()` method. The Connection Tracking layer is discussed in depth in Chapter 9, which deals with the netfilter subsystem.

- `int skb_iif`

  The `ifindex` of the network device on which the packet arrived.

- `__u32 rxhash`

  The `rxhash` of the SKB is calculated in the receive path, according to the source and destination address of the IP header and the ports from the transport header. A value of zero indicates that the hash is not valid. The `rxhash` is used to ensure that packets with the same flow will be handled by the same CPU when working with Symmetrical Multiprocessing (SMP). This decreases the number of cache misses and improves network performance. The `rxhash` is part of the Receive Packet Steering (RPS) feature, which was contributed by Google developers (Tom Herbert and others). The RPS feature gives performance improvement in SMP environments. See more in `Documentation/networking/scaling.txt`.

- `__be16 vlan_proto`

  The VLAN protocol used—usually it is the `802.1q` protocol. Recently support for the `802.1ad` protocol (also known as Stacked VLAN) was added.

  The following is an example of creating `802.1q` and `802.1ad` VLAN devices in userspace using the `ip` command of the `iproute2` package:

```
ip link add link eth0 eth0.1000 type vlan proto 802.1ad id 1000
ip link add link eth0.1000 eth0.1000.1000 type vlan proto 802.1q id 100
```

  Note: this feature is supported in kernel 3.10 and higher.

- `__u16 vlan_tci`

  The VLAN tag control information (2 bytes), composed of ID and priority.

Helper method:

- vlan_tx_tag_present(__skb): This macro checks whether the VLAN_TAG_PRESENT flag is set in the vlan_tci field of the specified __skb.

- __u16 queue_mapping

  Queue mapping for multiqueue devices.

  Helper methods:

  - skb_set_queue_mapping (struct sk_buff *skb, u16 queue_mapping): Sets the specified queue_mapping for the specified skb.
  - skb_get_queue_mapping(const struct sk_buff *skb): Returns the queue_mapping of the specified skb.

- __u8 pfmemalloc

  Allocate the SKB from PFMEMALLOC reserves.

  Helper method:

  - skb_pfmemalloc(): Returns true if the SKB was allocated from PFMEMALLOC reserves.

- __u8 ooo_okay:1

  The ooo_okay flag is set to avoid ooo (out of order) packets.

- __u8 l4_rxhash:1

  A flag that is set when a canonical 4-tuple hash over transport ports is used.

  See the __skb_get_rxhash() method in net/core/flow_dissector.c.

- __u8 no_fcs:1

  A flag that is set when you request the NIC to treat the last 4 bytes as Ethernet Frame Check Sequence (FCS).

- __u8 encapsulation:1

  The encapsulation field denotes that the SKB is used for encapsulation. It is used, for example, in the VXLAN driver. VXLAN is a standard protocol to transfer Layer 2 Ethernet packets over a UDP kernel socket. It can be used as a solution when there are firewalls that block tunnels and allow, for example, only TCP or UDP traffic. The VXLAN driver uses UDP encapsulation and sets the SKB encapsulation to 1 in the vxlan_init_net() method. Also the ip_gre module and the ipip tunnel module use encapsulation and set the SKB encapsulation to 1.

- __u32 secmark

  Security mark field. The secmark field is set by an iptables SECMARK target, which labels packets with any valid security context. For example:

```
iptables -t mangle -A INPUT -p tcp --dport 80 -j SECMARK --selctx
system_u:object_r:httpd_packet_t:s0
iptables -t mangle -A OUTPUT -p tcp --sport 80 -j SECMARK --selctx
system_u:object_r:httpd_packet_t:s0
```

  In the preceding rule, you are statically labeling packets arriving at and leaving from port 80 as httpd_packet_t. See: netfilter/xt_SECMARK.c.

Helper methods:

- void skb_copy_secmark(struct sk_buff *to, const struct sk_buff *from): Sets the value of the secmark field of the first specified SKB (to) to be equal to the value of the secmark field of the second specified SKB (from).
- void skb_init_secmark(struct sk_buff *skb): Initializes the secmark of the specified skb to be 0.

The next three fields: mark, dropcount, and reserved_tailroom appear in a union.

- __u32 mark

  This field enables identifying the SKB by marking it.

  You can set the mark field of the SKB, for example, with the iptables MARK target in an iptables PREROUTING rule with the mangle table.

- iptables -A PREROUTING -t mangle -i eth1 -j MARK --set-mark   0x1234

  This rule will assign the value of 0x1234 to every SKB mark field for incoming traffic on eth1 before performing a routing lookup. You can also run an iptables rule which will check the mark field of every SKB to match a specified value and act upon it. Netfilter targets and iptables are discussed in Chapter 9, which deals with the netfilter subsystem.

- __u32 dropcount

  The dropcount counter represents the number of dropped packets (sk_drops) of the sk_receive_queue of the assigned sock object (sk). See the sock_queue_rcv_skb() method in net/core/sock.c.

- _u32 reserved_tailroom: Used in the sk_stream_alloc_skb() method.
- sk_buff_data_t transport_header

  The transport layer (L4) header.

  Helper methods:

  - skb_transport_header(const struct sk_buff *skb): Returns the transport header of the specified skb.
  - skb_transport_header_was_set(const struct sk_buff *skb): Returns 1 if the transport_header of the specified skb is set.

- sk_buff_data_t network_header

  The network layer (L3) header.

  Helper method:

  - skb_network_header(const struct sk_buff *skb): Returns the network header of the specified skb.

- sk_buff_data_t mac_header

  The link layer (L2) header.

Helper methods:

- skb_mac_header(const struct sk_buff *skb): Returns the MAC header of the specified skb.
- skb_mac_header_was_set(const struct sk_buff *skb): Returns 1 if the mac_header of the specified skb was set.

- sk_buff_data_t tail

  The tail of the data.

- sk_buff_data_t end

  The end of the buffer. The tail cannot exceed end.

- unsigned char head

  The head of the buffer.

- unsigned char data

  The data head. The data block is allocated separately from the sk_buff allocation.

  See, in _alloc_skb(), net/core/skbuff.c:

```
data = kmalloc_reserve(size, gfp_mask, node, &pfmemalloc);
```

  Helper methods:

  - skb_headroom(const struct sk_buff *skb): This method returns the headroom, which is the number of bytes of free space at the head of the specified skb (skb->data – skb->head). See Figure A-1.
  - skb_tailroom(const struct sk_buff *skb): This method returns the tailroom, which is the number of bytes of free space at the tail of the specified skb (skb->end – skb->tail). See Figure A-1.

Figure A-1 shows the headroom and the tailroom of an SKB.

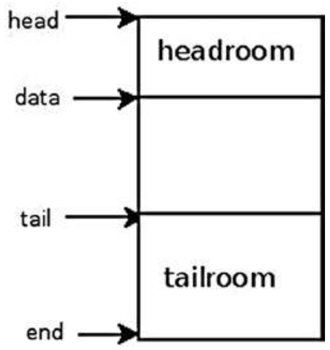

***Figure A-1.** Headroom and tailroom of an SKB*

The following are some methods for handling buffers:

- `skb_put(struct sk_buff *skb, unsigned int len)`: Adds data to a buffer: this method adds `len` bytes to the buffer of the specified `skb` and increments the length of the specified `skb` by the specified `len`.
- `skb_push(struct sk_buff *skb, unsigned int len)`: Adds data to the start of a buffer; this method decrements the data pointer of the specified `skb` by the specified `len` and increments the length of the specified `skb` by the specified `len`.
- `skb_pull(struct sk_buff *skb, unsigned int len)`: Removes data from the start of a buffer; this method increments the data pointer of the specified `skb` by the specified `len` and decrements the length of the specified `skb` by the specified `len`.
- `skb_reserve(struct sk_buff *skb, int len)`: Increases the headroom of an empty `skb` by reducing the tail.

After describing some methods for handling buffers, I continue with listing the members of the `sk_buff` structure:

- `unsigned int truesize`

  The total memory allocated for the SKB (including the SKB structure itself and the size of the allocated data block).

- `atomic_t users`

  A reference counter, initialized to 1; incremented by the `skb_get()` method and decremented by the `kfree_skb()` method or by the `consume_skb()` method; the `kfree_skb()` method decrements the usage counter; if it reached 0, the method will free the SKB—otherwise, the method will return without freeing it.

  Helper methods:

  - `skb_get(struct sk_buff *skb)`: Increments the `users` reference counter by 1.
  - `skb_shared(const struct sk_buff *skb)`: Returns `true` if the number of `users` is not 1.
  - `skb_share_check(struct sk_buff *skb, gfp_t pri)`: If the buffer is not shared, the original buffer is returned. If the buffer is shared, the buffer is cloned, and the old copy drops a reference. A new clone with a single reference is returned. When being called from interrupt context or with spinlocks held, the `pri` parameter (priority) must be GFP_ATOMIC. If memory allocation fails, NULL is returned.
  - `consume_skb(struct sk_buff *skb)`: Decrements the `users` reference counter and frees the SKB if the `users` reference counter is zero.

struct skb_shared_info

The `skb_shared_info struct` is located at the end of the data block (`skb_end_pointer(SKB)`). It consists of only a few fields. Let's take a look at it:

```
struct skb_shared_info {
    unsigned char       nr_frags;
    __u8                tx_flags;
    unsigned short      gso_size;
    unsigned short      gso_segs;
    unsigned short      gso_type;
    struct sk_buff          *frag_list;
    struct skb_shared_hwtstamps hwtstamps;
    __be32                  ip6_frag_id;
    atomic_t                dataref;
    void *                  destructor_arg;
    skb_frag_t              frags[MAX_SKB_FRAGS];
};
```

The following is a description of some of the important members of the skb_shared_info structure:

- nr_frags: Represents the number of elements in the frags array.
- tx_flags can be:
    - SKBTX_HW_TSTAMP: Generate a hardware time stamp.
    - SKBTX_SW_TSTAMP: Generate a software time stamp.
    - SKBTX_IN_PROGRESS: Device driver is going to provide a hardware timestamp.
    - SKBTX_DEV_ZEROCOPY: Device driver supports Tx zero-copy buffers.
    - SKBTX_WIFI_STATUS: Generate WiFi status information.
    - SKBTX_SHARED_FRAG: Indication that at least one fragment might be overwritten.
- When working with fragmentation, there are cases when you work with a list of sk_buffs (frag_list), and there are cases when you work with the frags array. It depends mostly on whether the Scatter/Gather mode is set.

  Helper methods:

    - skb_is_gso(const struct sk_buff *skb): Returns true if the gso_size of the skb_shared_info associated with the specified skb is not 0.
    - skb_is_gso_v6(const struct sk_buff *skb): Returns true if the gso_type of the skb_shared_info associated with the skb is SKB_GSO_TCPV6.
    - skb_shinfo(skb): A macro that returns the skb_shinfo associated with the specified skb.
    - skb_has_frag_list(const struct sk_buff *skb): Returns true if the frag_list of the skb_shared_info of the specified skb is not NULL.
    - dataref: A reference counter of the skb_shared_info struct. It is set to 1 in the method, which allocates the skb and initializes skb_shared_info (The __alloc_skb() method).

The net_device structure

The net_device struct represents the network device. It can be a physical device, like an Ethernet device, or it can be a software device, like a bridge device or a VLAN device. As with the sk_buff structure, I will list its important members. The net_device struct is defined in include/linux/netdevice.h:

- char name[IFNAMSIZ]

  The name of the network device. This is the name that you see with ifconfig or ip commands (for example eth0, eth1, and so on). The maximum length of the interface name is 16 characters. In newer distributions with biosdevname support, the naming scheme corresponds to the physical location of the network device. So PCI network devices are named p<slot>p<port>, according to the chassis labels, and embedded ports (on motherboard interfaces) are named em<port>—for example, em1, em2, and so on. There is a special suffix for SR-IOV devices and Network Partitioning (NPAR)-enabled devices. Biosdevname is developed by Dell: http://linux.dell.com/biosdevname. See also this white paper: http://linux.dell.com/files/whitepapers/consistent_network_device_naming_in_linux.pdf.

Helper method:

- dev_valid_name(const char *name): Checks the validity of the specified network device name. A network device name must obey certain restrictions in order to enable creating corresponding sysfs entries. For example, it cannot be " . " or " .. "; its length should not exceed 16 characters. Changing the interface name can be done like this, for example: ip link set <oldDeviceName> p2p1 <newDeviceName>. So, for example, ip link set p2p1 name a12345678901234567 will fail with this message: Error: argument "a12345678901234567" is wrong: "name" too long. The reason is that you tried to set a device name that is longer than 16 characters. And running ip link set p2p1 name. will fail with RTNETLINK answers: Invalid argument, since you tried to set the device name to be ".", which is an invalid value. See dev_valid_name() in net/core/dev.c.

- struct hlist_node name_hlist

  This is a hash table of network devices, indexed by the network device name. A lookup in this hash table is performed by dev_get_by_name(). Insertion into this hash table is performed by the list_netdevice() method, and removal from this hash table is done with the unlist_netdevice() method.

- char *ifalias

  SNMP alias interface name. Its length can be up to 256 (IFALIASZ).

  You can create an alias to a network device using this command line:

```
ip link set <devName> alias myalias
```

  The ifalias name is exported via sysfs by /sys/class/net/<devName>/ifalias.

  Helper method:

  - dev_set_alias(struct net_device *dev, const char *alias, size_t len): Sets the specified alias to the specified network device. The specified len parameter is the number of bytes of specified alias to be copied; if the specified len is greater than 256 (IFALIASZ), the method will fail with -EINVAL.

- unsigned int irq

  The Interrupt Request (IRQ) number of the device. The network driver should call request_irq() to register itself with this IRQ number. Typically this is done in the probe() callback of the network device driver. The prototype of the request_irq() method is: int request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags, const char *name, void *dev). The first argument is the IRQ number. The sepcified handler is the Interrupt Service Routine (ISR). The network driver should call the free_irq() method when it no longer uses this irq. In many cases, this irq is shared (the request_irq() method is called with the IRQF_SHARED flag). You can view the number of interrupts that occurred on each core by running cat /proc/interrupts. You can set the SMP affinity of the irq by echo irqMask > /proc/irq/<irqNumber>/smp_affinity.

In an SMP machine, setting the SMP affinity of interrupts means setting which cores are allowed to handle the interrupt. Some PCI network interfaces use Message Signaled Interrupts (MSIs). PCI MSI interrupts are never shared, so the IRQF_SHARED flag is not set when calling the `request_irq()` method in these network drivers. See more info in `Documentation/PCI/MSI-HOWTO.txt`.

- `unsigned long state`

  A flag that can be one of these values:

  - __LINK_STATE_START: This flag is set when the device is brought up, by the `dev_open()` method, and is cleared when the device is brought down.
  - __LINK_STATE_PRESENT: This flag is set in device registration, by the `register_netdevice()` method, and is cleared in the `netif_device_detach()` method.
  - __LINK_STATE_NOCARRIER: This flag shows whether the device detected loss of carrier. It is set by the `netif_carrier_off()` method and cleared by the `netif_carrier_on()` method. It is exported by `sysfs` via `/sys/class/net/<devName>/carrier`.
  - __LINK_STATE_LINKWATCH_PENDING: This flag is set by the `linkwatch_fire_event()` method and cleared by the `linkwatch_do_dev()` method.
  - __LINK_STATE_DORMANT: The dormant state indicates that the interface is not able to pass packets (that is, it is not "up"); however, this is a "pending" state, waiting for some external event. See section 3.1.12, "New states for IfOperStatus" in RFC 2863, "The Interfaces Group MIB."

  The `state` flag can be set with the generic `set_bit()` method.

  Helper methods:

  - `netif_running(const struct net_device *dev)`: Returns `true` if the __LINK_STATE_START flag of the state field of the specified device is set.
  - `netif_device_present(struct net_device *dev)`: Returns `true` if the __LINK_STATE_PRESENT flag of the state field of the specified device is set.
  - `netif_carrier_ok (const struct net_device *dev)`: Returns `true` if the __LINK_STATE_NOCARRIER flag of the state field of the specified device is not set.

  These three methods are defined in `include/linux/netdevice.h`.

- `netdev_features_t features`

  The set of currently active device features. These features should be changed only by the network core or in error paths of the `ndo_set_features()` callback. Network driver developers are responsible for setting the initial set of the device features. Sometimes they can use a wrong combination of features. The network core fixes this by removing an offending feature in the `netdev_fix_features()` method, which is invoked when the network interface is registered (in the `register_netdevice()` method); a proper message is also written to the kernel log.

I will mention some net_device features here and discuss them. For the full list of net_device features, look in include/linux/netdev_features.h.

- NETIF_F_IP_CSUM means that the network device can checksum L4 IPv4 TCP/UDP packets.
- NETIF_F_IPV6_CSUM means that the network device can checksum L4 IPv6 TCP/UDP packets.
- NETIF_F_HW_CSUM means that the device can checksum in hardware all L4 packets. You cannot activate NETIF_F_HW_CSUM together with NETIF_F_IP_CSUM, or together with NETIF_F_IPV6_CSUM, because that will cause duplicate checksumming.

If the driver features set includes both NETIF_F_HW_CSUM and NETIF_F_IP_CSUM features, then you will get a kernel message saying "mixed HW and IP checksum settings." In such a case, the netdev_fix_features() method removes the NETIF_F_IP_CSUM feature. If the driver features set includes both NETIF_F_HW_CSUM and NETIF_F_IPV6_CSUM features, you get again the same message as in the previous case. This time, the NETIF_F_IPV6_CSUM feature is the one which is being removed by the netdev_fix_features() method. In order for a device to support TSO (TCP Segmentation Offload), it needs also to support Scatter/Gather and TCP checksum; this means that both NETIF_F_SG and NETIF_F_IP_CSUM features must be set. If the driver features set does not include the NETIF_F_SG feature, then you will get a kernel message saying "Dropping TSO features since no SG feature," and the NETIF_F_ALL_TSO feature will be removed. If the driver features set does not include the NETIF_F_IP_CSUM feature and does not include NETIF_F_HW_CSUM, then you will get a kernel message saying "Dropping TSO features since no CSUM feature," and the NETIF_F_TSO will be removed.

■ **Note** In recent kernels, if CONFIG_DYNAMIC_DEBUG kernel config item is set, you might need to explicitly enable printing of some messages, via <debugfs>/dynamic_debug/control interface. See Documentation/dynamic-debug-howto.txt.

- NETIF_F_LLTX is the LockLess TX flag and is considered deprecated. When it is set, you don't use the generic Tx lock (This is why it is called LockLess TX). See the following macro (HARD_TX_LOCK) from net/core/dev.c:

```
#define HARD_TX_LOCK(dev, txq, cpu) { \ if ((dev->features & NETIF_F_LLTX) == 0) { \
    __netif_tx_lock(txq, cpu); \
  } \
   }
```

NETIF_F_LLTX is used in tunnel drivers like VXLAN, VETH, and in IP over IP (IPIP) tunneling driver. For example, in the IPIP tunnel module, you set the NETIF_F_LLTX flag in the ipip_tunnel_setup() method (net/ipv4/ipip.c).

The NETIF_F_LLTX flag is also used in a few drivers that have implemented their own Tx lock, like the cxgb network driver.

In drivers/net/ethernet/chelsio/cxgb/cxgb2.c, you have:

```
static int __devinit init_one(struct pci_dev *pdev,
const struct pci_device_id *ent)
{
    . . .
    netdev->features |= NETIF_F_SG | NETIF_F_IP_CSUM |
                        NETIF_F_RXCSUM | NETIF_F_LLTX;
   . . .
}
```

- NETIF_F_GRO is used to indicate that the device supports GRO (Generic Receive Offload). With GRO, incoming packets are merged at reception time. The GRO feature improves network performance. GRO replaced LRO (Large Receive Offload), which was limited to TCP/IPv4. This flag is checked in the beginning of the dev_gro_receive() method; devices that do not have this flag set will not perform the GRO handling part in this method. A driver that wants to use GRO should call the napi_gro_receive() method in the Rx path of the driver. You can enable/disable GRO with ethtool, by ethtool -K <deviceName> gro on/ ethtool -K <deviceName> gro off, respectively. You can check whether GRO is set by running ethtool -k <deviceName> and looking at the gro field.
- NETIF_F_GSO is set to indicate that the device supports Generic Segmentation Offload (GSO). GSO is a generalization of a previous solution called TSO (TCP segmentation offload), which dealt only with TCP in IPv4. GSO can handle also IPv6, UDP, and other protocols. GSO is a performance optimization, based on traversing the networking stack once instead of many times, for big packets. So the idea is to avoid segmentation in Layer 4 and defer segmentation as much as possible. The sysadmin can enable/disable GSO with ethtool, by ethtool -K <driverName> gso on/ethtool -K <driverName> gso off, respectively. You can check whether GSO is set by running ethtool -k <deviceName> and looking at the gso field. To work with GSO, you should work in Scatter/Gather mode. The NETIF_F_SG flag must be set.
- NETIF_F_NETNS_LOCAL is set for network namespace local devices. These are network devices that are not allowed to move between network namespaces. The loopback, VXLAN, and PPP network devices are examples of namespace local devices. All these devices have the NETIF_F_NETNS_LOCAL flag set. A sysadmin can check whether an interface has the NETIF_F_NETNS_LOCAL flag set or not by ethtool -k <deviceName>. This feature is fixed and cannot be changed by ethtool. Trying to move a network device of this type to a different namespace results in an error (-EINVAL). For details, look in the dev_change_net_namespace() method (net/core/dev.c). When deleting a network namespace, devices that do not have the NETIF_F_NETNS_LOCAL flag set are moved to the default initial network namespace (init_net). Network namespace local devices that have the NETIF_F_NETNS_LOCAL flag set are not moved to the default initial network namespace (init_net), but are deleted.
- NETIF_F_HW_VLAN_CTAG_RX is for use by devices which support VLAN Rx hardware acceleration. It was formerly called NETIF_F_HW_VLAN_RX and was renamed in kernel 3.10, when support for 802.1ad was added. "CTAG" was added to indicate that this device differ from "STAG" device (Service provider tagging). A device driver that sets the NETIF_F_HW_VLAN_RX feature must also define the ndo_vlan_rx_add_vid() and ndo_vlan_rx_kill_vid() callbacks. Failure to do so will avoid device registration and result in a "Buggy VLAN acceleration in driver" kernel error message.

- NETIF_F_HW_VLAN_CTAG_TX is for use by devices that support VLAN Tx hardware acceleration. It was formerly called NETIF_F_HW_VLAN_TX and was renamed in kernel 3.10 when support for `802.1ad` was added.
- NETIF_F_VLAN_CHALLENGED is set for devices that can't handle VLAN packets. Setting this feature avoids registration of a VLAN device. Let's take a look at the VLAN registration method:

  ```
  static int register_vlan_device(struct net_device *real_dev, u16 vlan_id) {
      int err;
      . . .
      err = vlan_check_real_dev(real_dev, vlan_id);
  ```

  The first thing the `vlan_check_real_dev()` method does is to check the network device features and return an error if the NETIF_F_VLAN_CHALLENGED feature is set:

  ```
  int vlan_check_real_dev(struct net_device *real_dev, u16 vlan_id)
  {
          const char *name = real_dev->name;

          if (real_dev->features & NETIF_F_VLAN_CHALLENGED) {
                  pr_info("VLANs not supported on %s\n", name);
                  return -EOPNOTSUPP;
          }
                  . . .
  }
  ```

For example, some types of Intel e100 network device drivers set the NETIF_F_VLAN_CHALLENGED feature (see `e100_probe()` in `drivers/net/ethernet/intel/e100.c`).

You can check whether the NETIF_F_VLAN_CHALLENGED is set by running `ethtool -k <deviceName>` and looking at the `vlan-challenged` field. This is a fixed value that you cannot change with the `ethtool` command.

- NETIF_F_SG is set when the network interface supports Scatter/Gather IO. You can enable and disable Scatter/Gather with `ethtool`, by `ethtool -K <deviceName> sg on`/ `ethtool -K <deviceName> sg off`, respectively. You can check whether Scatter/Gather is set by running `ethtool -k <deviceName>` and looking at the `sg` field.
- NETIF_F_HIGHDMA is set if the device can perform access by DMA to high memory. The practical implication of setting this feature is that the `ndo_start_xmit()` callback of the `net_device_ops` object can manage SKBs, which have `frags` elements in high memory. You can check whether the NETIF_F_HIGHDMA is set by running `ethtool -k <deviceName>` and looking at the `highdma` field. This is a fixed value that you cannot change with the `ethtool` command.
- `netdev_features_t hw_features`

  The set of features that are changeable features. This means that their state may possibly be changed (enabled or disabled) for a particular device by a user's request. This set should be initialized in the `ndo_init()` callback and not changed later.

- `netdev_features_t wanted_features`

  The set of features that were requested by the user. A user may request to change various offloading features—for example, by running `ethtool -K eth1 rx on`. This generates a feature change event notification (NETDEV_FEAT_CHANGE) to be sent by the `netdev_features_change()` method.

- `netdev_features_t vlan_features`

  The set of features whose state is inherited by child VLAN devices. For example, let's look at the `rtl_init_one()` method, which is the `probe` callback of the `r8169` network device driver (see Chapter 14):

```
int rtl_init_one(struct pci_dev *pdev, const struct pci_device_id *ent)

{
   . . .
   dev->vlan_features=NETIF_F_SG|NETIF_F_IP_CSUM|NETIF_F_TSO|   NETIF_F_HIGHDMA;
   . . .
}
```

  (`drivers/net/ethernet/realtek/r8169.c`)

  This initialization means that all child VLAN devices will have these features. For example, let's say that your `eth0` device is an `r8169` device, and you add a VLAN device thus: `vconfig add eth0 100`. Then, in the initialization in the VLAN module, there is this code related to `vlan_features`:

```
static int vlan_dev_init(struct net_device *dev)
{
    . . .
    dev->features |= real_dev->vlan_features | NETIF_F_LLTX;
    . . .
}
```

  (`net/8021q/vlan_dev.c`)

  This means that it sets the features of the VLAN child device to be the `vlan_features` of the real device (which is `eth0` in this case), which were set according to what you saw earlier in the `rtl_init_one()` method.

- `netdev_features_t hw_enc_features`

  The mask of features inherited by encapsulating devices. This field indicates what encapsulation offloads the hardware is capable of doing, and drivers will need to set them appropriately. For more info about the network device features, see `Documentation/networking/netdev-features.txt`.

- `ifindex`

  The `ifindex` (Interface index) is a unique device identifier. This index is incremented by 1 each time you create a new network device, by the `dev_new_index()` method. The first network device you create, which is almost always the loopback device, has `ifindex` of 1. Cyclic integer overflow is handled by the method that handles assignment of the `ifindex` number. The `ifindex` is exported by `sysfs` via `/sys/class/net/<devName>/ifindex`.

- `struct net_device_stats stats`

  The statistics `struct`, which was left as a legacy, includes fields like the number of `rx_packets` or the number of `tx_packets`. New device drivers use the `rtnl_link_stats64 struct` (defined in `include/uapi/linux/if_link.h`) instead of the `net_device_stats struct`. Most of the network drivers implement the `ndo_get_stats64()` callback of `net_device_ops` (or the `ndo_get_stats()` callback of `net_device_ops`, when working with the older API).

  The statistics are exported via `/sys/class/net/<deviceName>/statistics`.

  Some drivers implement the `get_ethtool_stats()` callback. These drivers show statistics by `ethtool -S <deviceName>`

  See, for example, the `rtl8169_get_ethtool_stats()` method in `drivers/net/ethernet/realtek/r8169.c`.

- `atomic_long_t rx_dropped`

  A counter of the number of packets that were dropped in the RX path by the core network stack. This counter should not be used by drivers. Do not confuse the `rx_dropped` field of the `sk_buff` with the `dropped` field of the `softnet_data struct`. The `softnet_data struct` represents a per-CPU object. They are not equivalent because the `rx_dropped` of the `sk_buff` might be incremented in several methods, whereas the `dropped` counter of `softnet_data` is incremented only by the `enqueue_to_backlog()` method (`net/core/dev.c`). The `dropped` counter of `softnet_data` is exported by `/proc/net/softnet_stat`. In `/proc/net/softnet_stat` you have one line per CPU. The first column is the total packets counter, and the second one is the dropped packets counter.

For example:

```
cat /proc/net/softnet_stat
00000076 00000001 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000005 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

You see here one line per CPU (you have two CPUs); for the first CPU, you see 118 total packets (hex 0x76), where one packet is dropped. For the second CPU, you see 5 total packets and 0 dropped.

- `struct net_device_ops *netdev_ops`

  The `netdev_ops` structure includes pointers for several callback methods that you want to define if you want to override the default behavior. Here are some callbacks of `netdev_ops`:

  - The `ndo_init()` callback is called when network device is registered.
  - The `ndo_uninit()` callback is called when the network device is unregistered or when the registration fails.
  - The `ndo_open()` callback handles change of device state, when a network device state is being changed from down state to up state.
  - The `ndo_stop()` callback is called when a network device state is being changed to be down.
  - The `ndo_validate_addr()` callback is called to check whether the MAC is valid. Many network drivers set the generic `eth_validate_addr()` method to be the `ndo_validate_addr()` callback. The generic `eth_validate_addr()` method returns `true` if the MAC address is not a multicast address and is not all zeroes.

- The ndo_set_mac_address() callback sets the MAC address. Many network drivers set the generic eth_mac_addr() method to be the ndo_set_mac_address() callback of struct net_device_ops for setting their MAC address. For example, the VETH driver (drivers/net/veth.c) or the VXLAN driver (drivers/nets/vxlan.c).
- The ndo_start_xmit() callback handles packet transmission. It cannot be NULL.
- The ndo_select_queue() callback is used to select a Tx queue, when working with multiqueues. If the ndo_select_queue() callback is not set, then the __netdev_pick_tx() is called. See the implementaion of the netdev_pick_tx() method in net/core/flow_dissector.c.
- The ndo_change_mtu() callback handles modifying the MTU. It should check that the specified MTU is not less than 68, which is the minimum MTU. In many cases, network drivers set the ndo_change_mtu() callback to be the generic eth_change_mtu() method. The eth_change_mtu() method should be overridden if jumbo frames are supported.
- The ndo_do_ioctl() callback is called when getting an IOCTL request which is not handled by the generic interface code.
- The ndo_tx_timeout() callback is called when the transmitter was idle for a quite a while (for watchdog usage).
- The ndo_add_slave() callback is called to set a specified network device as a slave to a specified netowrk device. It is used, for example, in the team network driver and in the bonding network driver.
- The ndo_del_slave() callback is called to remove a previously enslaved network device.
- The ndo_set_features() callback is called to update the configuration of a network device with new features.
- The ndo_vlan_rx_add_vid() callback is called when registering a VLAN id if the network device supports VLAN filtering (the NETIF_F_HW_VLAN_FILTER flag is set in the device features).
- The ndo_vlan_rx_kill_vid() callback is called when unregistering a VLAN id if the network device supports VLAN filtering (the NETIF_F_HW_VLAN_FILTER flag is set in the device features).

■ **Note** From kernel 3.10, the NETIF_F_HW_VLAN_FILTER flag was renamed to NETIF_F_HW_VLAN_CTAG_FILTER.

- There are also several callbacks for handling SR-IOV devices, for example, ndo_set_vf_mac() and ndo_set_vf_vlan().

Before kernel 2.6.29, there was a callback named set_multicast_list() for addition of multicast addresses, which was replaced by the dev_set_rx_mode() method. The dev_set_rx_mode() callback is called primarily whenever the unicast or multicast address lists or the network interface flags are updated.

- struct ethtool_ops *ethtool_ops

The ethtool_ops structure includes pointers for several callbacks for handling offloads, getting and setting various device settings, reading registers, getting statistics, reading RX flow hash indirection table, WakeOnLAN parameters, and many more. If the network driver does not initialize the ethtool_ops object, the networking core provides a default empty `ethtool_ops` object named `default_ethtool_ops`. The management of `ethtool_ops` is done in `net/core/ethtool.c`.

Helper method:

- SET_ETHTOOL_OPS (netdev,ops): A macro which sets the specified `ethtool_ops` for the specified `net_device`.

You can view the offload parameters of a network interface device by running `ethtool -k <deviceName>`. You can set some offload parameters of a network interface device by running `ethtool -K <deviceName> offloadParameter off/on`. See `man 8 ethtool`.

- `const struct header_ops *header_ops`

  The `header_ops struct` include callbacks for creating the Layer 2 header, parsing it, rebuilding it, and more. For Ethernet it is `eth_header_ops`, defined in `net/ethernet/eth.c`.

- `unsigned int flags`

  The interface flags of the network device that you can see from userspace. Here are some flags (for a full list see `include/uapi/linux/if.h`):

  - IFF_UP flag is set when the interface state is changed from down to up.
  - IFF_PROMISC is set when the interface is in promiscuous mode (receives all packets). When running sniffers like `wireshark` or `tcpdump`, the network interface is in promiscuous mode.
  - IFF_LOOPBACK is set for the loopback device.
  - IFF_NOARP is set for devices which do not use the ARP protocol. IFF_NOARP is set, for example, in tunnel devices (see for example, in the `ipip_tunnel_setup()` method, `net/ipv4/ipip.c`).
  - IFF_POINTOPOINT is set for PPP devices. See for example, the `ppp_setup()` method, `drivers/net/ppp/ppp_generic.c`.
  - IFF_MASTER is set for master devices. See, for example, for bonding devices, the `bond_setup()` method in `drivers/net/bonding/bond_main.c`.
  - IFF_LIVE_ADDR_CHANGE flag indicates that the device supports hardware address modification when it’s running. See the `eth_mac_addr()` method in `net/ethernet/eth.c`.
  - IFF_UNICAST_FLT flag is set when the network driver handles unicast address filtering.
  - IFF_BONDING is set for a bonding master device or bonding slave device. The bonding driver provides a method for aggregating multiple network interfaces into a single logical interface.
  - IFF_TEAM_PORT is set for a device used as a team port. The teaming driver is a load-balancing network software driver intended to replace the bonding driver.
  - IFF_MACVLAN_PORT is set for a device used as a macvlan port.
  - IFF_EBRIDGE is set for an Ethernet bridging device.

  The `flags` field is exported by `sysfs` via `/sys/class/net/<devName>/flags`.

Some of these flags can be set by userspace tools. For example, `ifconfig <deviceName> -arp` will set the IFF_NOARP network interface flag, and `ifconfig <deviceName> arp` will clear the IFF_NOARP flag. Note that you can do the same with the `iproute2` `ip` command: `ip link set dev <deviceName> arp on` and `ip link set dev <deviceName> arp off`.

- `unsigned int priv_flags`

  The interface flags, which are invisible from userspace. For example, IFF_EBRIDGE for a bridge interface or IFF_BONDING for a bonding interface, or IFF_SUPP_NOFCS for an interface support sending custom FCS.

  Helper methods:

  - `netif_supports_nofcs()`: Returns `true` if the IFF_SUPP_NOFCS is set in the `priv_flags` of the specified device.
  - `is_vlan_dev(struct net_device *dev)`: Returns 1 if the IFF_802_1Q_VLAN flag is set in the `priv_flags` of the specified network device.

- `unsigned short gflags`

  Global flags (kept as legacy).

- `unsigned short padded`

  How much padding is added by the `alloc_netdev()` method.

- `unsigned char operstate`

  RFC 2863 operstate.

- `unsigned char link_mode`

  Mapping policy to operstate.

- `unsigned int mtu`

  The network interface MTU (Maximum Transmission Unit) value. The maximum size of frame the device can handle. RFC 791 sets 68 as a minimum MTU. Each protocol has MTU of its own. The default MTU for Ethernet is 1,500 bytes. It is set in the `ether_setup()` method, `net/ethernet/eth.c`. Ethernet packets with sizes higher than 1,500 bytes, up to 9,000 bytes, are called Jumbo frames. The network interface MTU is exported by `sysfs` via `/sys/class/net/<devName>/mtu.`

  Helper method:

  - `dev_set_mtu(struct net_device *dev, int new_mtu)`: Changes the MTU of the specified device to a new value, specified by the `mtu` parameter.

  The sysadmin can change the MTU of a network interface to 1,400, for example, in one of the following ways:

```
ifconfig <netDevice> mtu 1400
ip link set <netDevice> mtu 1400
echo 1400 > /sys/class/net/<netDevice>/mtu
```

  Many drivers implement the `ndo_change_mtu()` callback to change the MTU to perform driver-specific needed actions (like resetting the network card).

- `unsigned short type`

  The network interface hardware type. For example, for Ethernet it is ARPHRD_ETHER and is set in `ether_setup()` in `net/ethernet/eth.c`. For PPP interface, it is ARPHRD_PPP, and is set in the `ppp_setup()` method in `drivers/net/ppp/ppp_generic.c`. The type is exported by `sysfs` via `/sys/class/net/<devName>/type`.

- `unsigned short hard_header_len`

  The hardware header length. Ethernet headers, for example, consist of MAC source address, MAC destination address, and a type. The MAC source and destination addresses are 6 bytes each, and the type is 2 bytes. So the Ethernet header length is 14 bytes. The Ethernet header length is set to 14 (ETH_HLEN) in the `ether_setup()` method, `net/ethernet/eth.c`. The `ether_setup()` method is responsible for initializing some Ethernet device defaults, like the hard header len, Tx queue len, MTU, type, and more.

- `unsigned char perm_addr[MAX_ADDR_LEN]`

  The permanent hardware address (MAC address) of the device.

- `unsigned char addr_assign_type`

  Hardware address assignment type, can be one of the following:

  - NET_ADDR_PERM
  - NET_ADDR_RANDOM
  - NET_ADDR_STOLEN
  - NET_ADDR_SET

    By default, the MAC address is permanent (NET_ADDR_PERM). If the MAC address was generated with a helper method named `eth_hw_addr_random()`, the type of the MAC address is NET_ADD_RANDOM. The type of the MAC address is stored in the `addr_assign_type` member of the `net_device`. Also when changing the MAC address of the device, with `eth_mac_addr()`, you reset the `addr_assign_type` with ~NET_ADDR_RANDOM (if it was marked as NET_ADDR_RANDOM before). When a network device is registered (by the `register_netdevice()` method), if the `addr_assign_type` equals NET_ADDR_PERM, `dev->perm_addr` is set to be `dev->dev_addr`. When you set a MAC address, you set the `addr_assign_type` to be NET_ADDR_SET. This indicates that the MAC address of a device has been set by the `dev_set_mac_address()` method. The `addr_assign_type` is exported by `sysfs` via `/sys/class/net/<devName>/addr_assign_type`.

- `unsigned char addr_len`

  The hardware address length in octets. For Ethernet addresses, it is 6 (ETH_ALEN) bytes and is set in the `ether_setup()` method. The `addr_len` is exported by `sysfs` via `/sys/class/net/<deviceName>/addr_len`.

- `unsigned char neigh_priv_len`

  Used in the `neigh_alloc()` method, `net/core/neighbour.c`; `neigh_priv_len` is initialized only in the ATM code (`atm/clip.c`).

- `struct netdev_hw_addr_list uc`

  Unicast MAC addresses list, initialized by the `dev_uc_init()` method. There are three types of packets in Ethernet: unicast, multicast, and broadcast. Unicast is destined for one machine, multicast is destined for a group of machines, and broadcast is destined for all the machines in the LAN.

  Helper methods:

  - `netdev_uc_empty(dev)`: Returns 1 if the unicast list of the specified device is empty (its `count` field is 0).
  - `dev_uc_flush(struct net_device *dev)`: Flushes the unicast addresses of the specified network device and zeroes `count`.

- `struct netdev_hw_addr_list mc`

  Multicast MAC addresses list, initialized by the `dev_mc_init()` method.

  Helper methods:

  - `netdev_mc_empty(dev)`: Returns 1 if the multicast list of the specified device is empty (its `count` field is 0).
  - `dev_mc_flush(struct net_device *dev)`: Flushes the multicast addresses of the specified network device and zeroes the count field.

- `unsigned int promiscuity`

  A counter of the times a network interface card is told to work in promiscuous mode. With promiscuous mode, packets with MAC destination address which is different than the interface MAC address are not rejected. The `promiscuity` counter is used, for example, to enable more than one sniffing client; so when opening some sniffing clients (like `wireshark`), this counter is incremented by 1 for each client you open, and closing that client will decrement the promiscuity counter. When the last instance of the sniffing client is closed, `promiscuity` will be set to 0, and the device will exit from working in promiscuous mode. It is used also in the bridging subsystem, as the bridge interface needs to work in promiscuous mode. So when adding a bridge interface, the network interface card is set to work in promiscuous mode. See the call to the `dev_set_promiscuity()` method in `br_add_if()`, `net/bridge/br_if.c`.

  Helper method:

  - `dev_set_promiscuity(struct net_device *dev, int inc)`: Increments/decrements the `promiscuity` counter of the specified network device according to the specified increment. The `dev_set_promiscuity()` method can get a positive increment or a negative increment parameter. As long as the promiscuity counter remains above zero, the interface remains in promiscuous mode. Once it reaches zero, the device reverts back to normal filtering operation. Because promiscuity is an integer, the `dev_set_promiscuity()` method takes into account cyclic overflow of integer, which means it handles the case when the `promiscuity` counter is incremented when it reaches the maximum positive value an unsigned integer can reach.

- `unsigned int allmulti`

  The `allmulti` counter of the network device enables or disables the allmulticast mode. When selected, all multicast packets on the network will be received by the interface. You can set a network device to work in allmulticast mode by `ifconfig eth0 allmulti`. You disable the `allmulti` flag by `ifconfig eth0 -allmulti`.

Enabling/disabling the allmulticast mode can also be performed with the `ip` command:

```
ip link set p2p1 allmulticast on
ip link set p2p1 allmulticast off
```

You can also see the allmulticast state by inspecting the flags that are shown by the `ip` command:

```
ip addr show
flags=4610<BROADCAST,ALLMULTI,MULTICAST>  mtu 1500
```

Helper method:

- `dev_set_allmulti(struct net_device *dev, int inc)`: Increments/decrements the `allmulti` counter of the specified network device according to the specified increment (which can be a positive or a negative integer). The `dev_set_allmulti()` method also sets the IFF_ALLMULTI flag of the network device when setting the allmulticast mode and removes this flag when disabling the allmulticast mode.

The next three fields are protocol-specific pointers:

- `struct in_device __rcu *ip_ptr`

  This pointer is assigned to a pointer to `struct in_device`, which represents IPv4 specific data, in `inetdev_init()`, `net/ipv4/devinet.c`.

- `struct inet6_dev __rcu *ip6_ptr`

  This pointer is assigned to a pointer to `struct inet6_dev`, which represents IPv6 specific data, in `ipv6_add_dev()`, `net/ipv6/addrconf.c`.

- `struct wireless_dev *ieee80211_ptr`

  This is a pointer for the wireless device, assigned in the `ieee80211_if_add()` method, `net/mac80211/iface.c`.

- `unsigned long last_rx`

  Time of last Rx. It should not be set by network device drivers, unless really needed. Used, for example, in the bonding driver code.

- `struct list_head dev_list`

  The global list of network devices. Insertion to the list is done with the `list_netdevice()` method, when the network device is registered. Removal from the list is done with the `unlist_netdevice()` method, when the network device is unregistered.

- `struct list_head napi_list`

  NAPI stands for New API, a technique by which the network driver works in polling mode, and not in interrupt-driven mode, when it is under high traffic. Using NAPI under high traffic has been proven to improve performance. When working with NAPI, instead of getting an interrupt for each received packet, the network stack buffers the packets and from time to time triggers the poll method the driver registered with the `netif_napi_add()` method. When working with polling mode, the driver starts to work in interrupt-driven mode. When there is an interrupt for the first received packet, you reach the interrupt service routine (ISR), which is the method that was registered with `request_irq()`. Then the driver disables interrupts and notifies NAPI to take control, usually by calling the __napi_schedule() method from the ISR. See, for example, the cpsw_interrupt() method in drivers/net/ethernet/ti/cpsw.

When the traffic is low, the network driver switches to work in interrupt-driven mode. Nowadays, most network drivers work with NAPI. The napi_list object is the list of napi_struct objects; The netif_napi_add() method adds napi_struct objects to this list, and the netif_napi_del() method deletes napi_struct objects from this list. When calling the netif_napi_add() method, the driver should specify its polling method and a weight parameter. The weight is a limit on the number of packets the driver will pass to the stack in each polling cycle. It is recommended to use a weight of 64. If a driver attempts to call netif_napi_add() with weight higher than 64 (NAPI_POLL_WEIGHT), there is a kernel error message. NAPI_POLL_WEIGHT is defined in include/linux/netdevice.h.

The network driver should call napi_enable() to enable NAPI scheduling. Usually this is done in the ndo_open() callback of the net_device_ops object. The network driver should call napi_disable() to disable NAPI scheduling. Usually this is done in the ndo_stop() callback of net_device_ops. NAPI is implemented using softirqs. This softirq handler is the net_rx_action() method and is registered by calling open_softirq(NET_RX_SOFTIRQ, net_rx_action) by the net_dev_init() method in net/core/dev.c. The net_rx_action() method invokes the poll method of the network driver which was registered with NAPI. The maximum number of packets (taken from all interfaces which are registered to polling) in one polling cycle (NAPI poll) is by default 300. It is the netdev_budget variable, defined in net/core/dev.c, and can be modified via a procfs entry, /proc/sys/net/core/netdev_budget. In the past, you could change the weight per device by writing values to a procfs entry, but currently, the /sys/class/net/<device>/weight sysfs entry is removed. See Documentation/sysctl/net.txt. I should also mention that the napi_complete() method removes a device from the polling list. When a network driver wants to return to work in interrupt-driven mode, it should call the napi_complete() method to remove itself from the polling list.

- struct list_head unreg_list

  The list of unregistered network devices. Devices are added to this list when they are unregistered.

- unsigned char *dev_addr

  The MAC address of the network interface. Sometimes you want to assign a random MAC address. You do that by calling the eth_hw_addr_random() method, which also sets the addr_assign_type to be NET_ADDR_RANDOM.

  The dev_addr field is exported by sysfs via /sys/class/net/<devName>/address.

  You can change dev_addr with userspace tools like ifconfig or ip of iproute2.

  Helper methods: Many times you invoke the following helper methods on Ethernet addresses in general and on dev_addr field of a network device in particular:

  - is_zero_ether_addr(const u8 *addr): Returns true if the address is all zeroes.
  - is_multicast_ether_addr(const u8 *addr): Returns true if the address is a multicast address. By definition the broadcast address is also a multicast address.
  - is_valid_ether_addr (const u8 *addr): Returns true if the specified MAC address is not 00:00:00:00:00:00, is not a multicast address, and is not a broadcast address (FF:FF:FF:FF:FF:FF).

- `struct netdev_hw_addr_list dev_addrs`

  The list of device hardware addresses.

- `unsigned char broadcast[MAX_ADDR_LEN]`

  The hardware broadcast address. For Ethernet devices, the broadcast address is initialized to 0XFFFFFF in the `ether_setup()` method, `net/ethernet/eth.c`. The broadcast address is exported by `sysfs` via `/sys/class/net/<devName>/broadcast`.

- `struct kset *queues_kset`

  A `kset` is a group of `kobjects` of a specific type, belonging to a specific subsystem.

  The `kobject` structure is the basic type of the device model. A Tx queue is represented by struct `netdev_queue`, and the Rx queue is represented by `struct netdev_rx_queue`. Each of them holds a `kobject` pointer. The `queues_kset` object is a group of all `kobjects` of the Tx queues and Rx queues. Each Rx queue has the `sysfs` entry `/sys/class/net/<deviceName>/queues/<rx-queueNumber>`, and each Tx queue has the `sysfs` entry `/sys/class/net/<deviceName>/queues/<tx-queueNumber>`. These entries are added with the `rx_queue_add_kobject()` method and the `netdev_queue_add_kobject()` method respectively, in `net/core/net-sysfs.c`. For more information about the `kobject` and the device model, see `Documentation/kobject.txt`.

- `struct netdev_rx_queue *_rx`

  An array of Rx queues (`netdev_rx_queue` objects), initialized by the `netif_alloc_rx_queues()` method. The Rx queue to be used is determined in the `get_rps_cpu()` method. See more info about RPS in the description of the `rxhash` field in the previous `sk_buff` section.

- `unsigned int num_rx_queues`

  The number of Rx queues allocated in the `register_netdev()` method.

- `unsigned int real_num_rx_queues`

  Number of Rx queues currently active in the device.

  Helper method:

  - `netif_set_real_num_rx_queues (struct net_device *dev, unsigned int rxq)`: Sets the actual number of Rx queues used for the specified device according to the specified number of Rx queues. The relevant `sysfs` entries (`/sys/class/net/<devName>/queues/*`) are updated (only in the case that the state of the device is NETREG_REGISTERED or NETREG_UNREGISTERING). Note that `alloc_netdev_mq()` initializes `num_rx_queues`, `real_num_rx_queues`, `num_tx_queues` and `real_num_tx_queues` to the same value. One can set the number of Tx queues and Rx queues by using `ip link` when adding a device. For example, if you want to create a VLAN device with 6 Tx queues and 7 Rx queues, you can run this command:

    ```
    ip link add link p2p1 name p2p1.1 numtxqueues 6 numrxqueues 7 type vlan id 8
    ```

  - `rx_handler_func_t __rcu *rx_handler`

Helper methods:

- `netdev_rx_handler_register(struct net_device *dev, rx_handler_func_t *rx_handler  void *rx_handler_data)`

The `rx_handler` callback is set by calling the `netdev_rx_handler_register()` method. It is used, for example, in bonding, team, openvswitch, macvlan, and bridge devices.

- `netdev_rx_handler_unregister(struct net_device *dev)`: Unregisters a receive handler for the specified network device.

- `void __rcu *rx_handler_data`

  The `rx_handler_data` field is also set by the `netdev_rx_handler_register()` method when a non-NULL value is passed to the `netdev_rx_handler_register()` method.

- `struct netdev_queue __rcu *ingress_queue`

  Helper method:

  - `struct netdev_queue *dev_ingress_queue(struct net_device *dev)`: Returns the `ingress_queue` of the specified `net_device` (`include/linux/rtnetlink.h`).

- `struct netdev_queue *_tx`

  An array of Tx queues (`netdev_queue` objects), initialized by the `netif_alloc_netdev_queues()` method.

  Helper method:

  - `netdev_get_tx_queue(const struct net_device *dev,unsigned int index)`: Returns the Tx queue (`netdev_queue` object), an element of the `_tx` array of the specified network device at the specified `index`.

- `unsigned int num_tx_queues`

  Number of Tx queues, allocated by the `alloc_netdev_mq()` method.

- `unsigned int real_num_tx_queues`

  Number of Tx queues currently active in the device.

  Helper method:

  - `netif_set_real_num_tx_queues(struct net_device *dev, unsigned int txq)`: Sets the actual number of Tx queues used.

- `struct Qdisc *qdisc`

  Each device maintains a queue of packets to be transmitted named `qdisc`. The Qdisc (Queuing Disciplines) layer implements the Linux kernel traffic management. The default `qdisc` is `pfifo_fast`. You can set a different `qdisc` using `tc`, the traffic control tool of the `iproute2` package. You can view the `qdisc` of your network device by the using the `ip` command:

```
ip addr show <deviceName>
```

For example, running

```
ip addr show eth1
```

can give:

```
2: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
link/ether 00:e0:4c:53:44:58 brd ff:ff:ff:ff:ff:ff
inet 192.168.2.200/24 brd 192.168.2.255 scope global eth1
inet6 fe80::2e0:4cff:fe53:4458/64 scope link
valid_lft forever preferred_lft forever
```

In this example, you can see that a `qdisc` of `pfifo_fast` is used, which is the default.

- `unsigned long tx_queue_len`

  The maximum number of allowed packets per queue. Each hardware layer has its own `tx_queue_len` default. For Ethernet devices, `tx_queue_len` is set to 1,000 by default (see the `ether_setup()` method). For FDDI, `tx_queue_len` is set to 100 by default (see the `fddi_setup()` method in `net/802/fddi.c`).

  The `tx_queue_len` field is set to 0 for virtual devices, such as the VLAN device, because the actual transmission of packets is done by the real device on which these virtual devices are based. You can set the Tx queue length of a device by using the command `ifconfig` (this option is called `txqueuelen`) or by using the command `ip link show` (it is called `qlen`), in this way, for example:

```
ifconfig  p2p1 txqueuelen 900
ip link set txqueuelen 950 dev p2p1
```

  The Tx queue length is exported via the following `sysfs` entry: `/sys/class/net/<deviceName>/tx_queue_len`.

- `unsigned long trans_start`

  The time (in `jiffies`) of the last transmission.

- `int watchdog_timeo`

  The watchdog is a timer that will invoke a callback when the network interface was idle and did not perform transmission in some specified timeout interval. Usually the driver defines a watchdog callback which will reset the network interface in such a case. The `ndo_tx_timeout()` callback of `net_device_ops` serves as the watchdog callback. The `watchdog_timeo` field represents the timeout that is used by the watchdog. See the `dev_watchdog()` method, `net/sched/sch_generic.c`.

- `int __percpu *pcpu_refcnt`

  Per CPU network device reference counter.

  Helper methods:

  - `dev_put(struct net_device *dev)`: Decrements the reference count.
  - `dev_hold(struct net_device *dev)`: Increments the reference count.

- `struct hlist_node index_hlist`

  This is a hash table of network devices, indexed by the network device index (the `ifindex` field). A lookup in this table is performed by the `dev_get_by_index()` method. Insertion into this table is performed by the `list_netdevice()` method, and removal from this list is done with the `unlist_netdevice()` method.

- `enum {...} reg_state`

  An `enum` that represents the various registration states of the network device.

  Possible values:

  - NETREG_UNINITIALIZED: When the device memory is allocated, in the `alloc_netdev_mqs()` method.
  - NETREG_REGISTERED: When the `net_device` is registered, in the `register_netdevice()` method.
  - NETREG_UNREGISTERING: When unregistering a device, in the `rollback_registered_many()` method.
  - NETREG_UNREGISTERED: The network device is unregistered but it is not freed yet.
  - NETREG_RELEASED: The network device is in the last stage of freeing the allocated memory of the network device, in the `free_netdev()` method.

- NETREG_DUMMY: Used in the `dummy` device, in the `init_dummy_netdev()` method. See `drivers/net/dummy.c`.

- `bool dismantle`

  A Boolean flag that shows that the device is in dismantle phase, which means that it is going to be freed.

- `enum {...} rtnl_link_state`

  This is an `enum` that can have two values that represent the two phases of creating a new link:

  - RTNL_LINK_INITIALIZE: The ongoing state, when creating the link is still not finished.
  - RTNL_LINK_INITIALIZING: The final state, when work is finished.

  See the `rtnl_newlink()` method in `net/core/rtnetlink.c`.

- `void (*destructor)(struct net_device *dev)`

  This destructor callback is called when unregistering a network device, in the `netdev_run_todo()` method. It enables network devices to perform additional tasks that need to be done for unregistering. For example, the loopback device destructor callback, `loopback_dev_free()`, calls `free_percpu()` for freeing its statistics object and `free_netdev()`. Likewise the team device destructor callback, `team_destructor()`, also calls `free_percpu()` for freeing its statistics object and `free_netdev()`. And there are many other network device drivers that define a `destructor` callback.

- `struct net *nd_net`

  The network namespace this network device is inside. Network namespaces support was added in the 2.6.29 kernel. These features provide process virtualization, which is considered lightweight in comparison to other virtualization solutions like KVM and Xen. There is currently support for six namespaces in the Linux kernel. In order to support network namespaces, a structure called `net` was added. This structure represents a network namespace. The process descriptor (`task_struct`) handles the network namespace and other namespaces via a new member which was added for namespaces support, named `nsproxy`. This `nsproxy` includes a network namespace object called `net_ns`, and also four other namespace objects of the following namespaces: pid namespace, mount namespace, uts namespace, and ipc namespace; the sixth namespace, the user namespace, is kept in struct `cred` (the credentials object) which is a member of the process descriptor, `task_struct`).

  Network namespaces provide a partitioning and isolation mechanism which enables one process or a group of processes to have a private view of a full network stack of their own. By default, after boot all network interfaces belong to the default network namespace, `init_net`. You can create a network namespace with userspace tools using the `ip` command from `iproute2` package or with the `unshare` command of `util-linux`—or by writing your own userspace application and invoking the `unshare()` or the `clone()` system calls with the CLONE_NEWNET flag. Moreover, you can also change the network namespace of a process by invoking the `setns()` system call. This `setns()` system call and the `unshare()` system call were added specially to support namespaces. The `setns()` system call can attach to the calling process an existing namespace of any type (network namespace, pid namespace, mount namespace, and so on). You need CAP_SYS_ADMIN privilege to call `set_ns()` for all namespaces, except the user namespace. See `man 2 setns`.

  A network device belongs to exactly one network namespace at a given moment. And a network socket belongs to exactly one network namespace at a given moment. Namespaces do not have names, but they do have a unique inode which identifies them. This unique inode is generated when the namespace is created and can be read by reading a `procfs` entry (the command `ls -al /proc/<pid>/ns/` shows all the unique inode numbers symbolic links of a process—you can also read these symbolic links with the `readlink` command).

  For example, using the `ip` command, creating a new namespace called `ns1` is done thus:

  ```
  ip netns add myns1
  ```

  Each newly created network namespace includes only the loopback device and includes no sockets. Each device (like a bridge device or a VLAN device) that is created from a process that runs in that namespace (like a shell) belongs to that namespace.

Removing a namespace is done using the following command:

```
ip netns del myns1
```

■ **Note** After deleting a namespace, all its physical network devices are moved to the default network namespace. Local devices (namespace local devices that have the NETIF_F_NETNS_LOCAL flag set, like PPP device or VXLAN device) are not moved to the default network namespace but are deleted.

Showing the list of all network namespaces on the system is done with this command:

```
ip netns list
```

Assigning the p2p1 interface to the `myns1` network namespace is done by the command:

```
ip link set p2p1 netns myns1
```

Opening a shell in `myns1` is done thus:

```
ip netns exec myns1 bash
```

With the `unshare` utility, creating a new namespace and starting a bash shell inside is done thus:

```
unshare --net bash
```

Two network namespaces can communicate by using a special virtual Ethernet driver, `veth.` (`drivers/net/veth.c`).

Helper methods:

- `dev_change_net_namespace(struct net_device *dev, struct net *net, const char *pat)`: Moves the network device to a different network namespace, specified by the `net` parameter. Local devices (devices in which the NETIF_F_NETNS_LOCAL feature is set) are not allowed to change their namespace. This method returns -EINVAL for this type of device. The `pat` parameter, when it is not NULL, is the name pattern to try if the current device name is already taken in the destination network namespace. The method also sends a KOBJ_REMOVE uevent for removing the old namespace entries from `sysfs`, and a KOBJ_ADD uevent to add the `sysfs` entries to the new namespace. This is done by invoking the `kobject_uevent()` method specifying the corresponding uevent.
- `dev_net(const struct net_device *dev)`: Returns the network namespace of the specified network device.
- `dev_net_set(struct net_device *dev, struct net *net)`: Decrements the reference count of the `nd_net` (namespace object) of the specified device and assigns the specified network namespace to it.

The following four fields are members in a union:

- `struct pcpu_lstats __percpu *lstats`

  The loopback network device statistics.
- `struct pcpu_tstats __percpu *tstats`

  The tunnel statistics.
- `struct pcpu_dstats __percpu *dstats`

  The dummy network device statistics.
- `struct pcpu_vstats __percpu *vstats`

  The VETH (Virtual Ethernet) statistics.
- `struct device dev`

The `device` object associated with the network device. Every device in the Linux kernel is associated with a device object, which is an instance of the `device` structure. For more information about the `device` structure, I suggest you read the "Devices" section in Chapter 14 of *Linux Device Drivers*, 3rd Edition (O'Reilly, 2005) and `Documentation/driver-model/overview.txt`.

Helper methods:

- `to_net_dev(d)`: Returns the `net_device` object that contains the specified device as its device object.
- SET_NETDEV_DEV (net, pdev): Sets the parent of the `dev` member of the specified network device to be that specified device (the second argument, `pdev`).

  With virtual devices, you do not call the SET_NETDEV_DEV() macro. As a result, entries for these virtual devices are created under `/sys/devices/virtual/net`.

  The SET_NETDEV_DEV() macro should be called before calling the `register_netdev()` method.
- SET_NETDEV_DEVTYPE(net, devtype): Sets the type of the `dev` member of the specified network device to be the specified type. The type is a `device_type` object.

  SET_NETDEV_DEVTYPE() is used, for example, in the `br_dev_setup()` method, in`net/bridge/br_device.c`:

```
static struct device_type br_type = {
.name = "bridge",
};

void br_dev_setup(struct net_device *dev)
{
    . . .
    SET_NETDEV_DEVTYPE(dev, &br_type);
    . . .

}
```

With the udevadm tool (udev management tool), you can find the device type, for example, for a bridge device named mybr:

```
udevadm info -q all -p /sys/devices/virtual/net/mybr

P: /devices/virtual/net/mybr

E: DEVPATH=/devices/virtual/net/mybr

E: DEVTYPE=bridge

E: ID_MM_CANDIDATE=1

E: IFINDEX=7

E: INTERFACE=mybr

E: SUBSYSTEM=net
```

- `const struct attribute_group *sysfs_groups[4]`

  Used by networking `sysfs`.

- `struct rtnl_link_ops *rtnl_link_ops`

  The rtnetlink link operations object. It consists of various callbacks for handling network devices, for example:

  - `newlink()` for configuring and registering a new device.
  - `changelink()` for changing parameters of an existing device.
  - `dellink()` for removing a device.
  - `get_num_tx_queues()` for getting the number of Tx queues.
  - `get_num_rx_queues()` for getting the number of Rx queues.

  Registration and unregistration of `rtnl_link_ops` object is done with the `rtnl_link_register()` method and the `rtnl_link_unregister()` method, respectively.

- `unsigned int gso_max_size`

  Helper method:

  - `netif_set_gso_max_size(struct net_device *dev, unsigned int size)`: Sets the specified `gso_max_size` for the specified network device.

- `u8 num_tc`

  The number of traffic classes in the net device.

Helper method:

- `netdev_set_num_tc(struct net_device *dev, u8 num_tc)`: Sets the `num_tc` of the specified network device (the maximum value of `num_tc` can be TC_MAX_QUEUE, which is 16).
- `int netdev_get_num_tc(struct net_device *dev)`: Returns the `num_tc` value of the specified network device.

- `struct netdev_tc_txq tc_to_txq[TC_MAX_QUEUE]`
- `u8 prio_tc_map[TC_BITMASK + 1];`
- `struct netprio_map __rcu *priomap`

  The network priority `cgroup` module provides an interface to set the priority of network traffic. The cgroups layer is a Linux kernel layer that enables process resource management and process isolation. It enables assigning one task or several tasks to a system resource, like a networking resource, memory resource, CPU resource, and so on. The cgroups layer implements a Virtual File System (VFS) and is managed by filesystem operations like mounting/unmounting, creating files and directories, writing to cgroup VFS control files, and so forth. The cgroup project was started in 2005 by developers from Google (Paul Manage, Rohit Seth, and others). Some projects are based on cgroups usage, like `systemd` and `lxc` (Linux containers). Google has its own implementation of containers, based on cgroups. There is no relation between the cgroup implementation and the namespaces implementation. In the past, there was a namespace controller in cgroups but it was removed. No new system calls were added for cgroups implementations, and the cgroup code additions are not critical in terms of performance. There are two networking cgroups modules: `net_prio` and `net_cls`. These two `cgroup` modules are relatively short and simple.

  Setting the priority of network traffic with the `netprio cgroup` module is done by writing an entry to a `cgroup` control file, `/sys/fs/cgroup/net_prio/<group>/net_prio.ifpriomap`. The entry is in the form "deviceName priority." It is true that an application can set the priority of its traffic via the `setsockopt()` system call with SO_PRIORITY, but this is not always possible. Sometimes you cannot change the code of certain applications. Moreover, you want to let the system administrator decide on priority according to site-specific setup. The `netprio` kernel module is a solution when using the `setsockopt()` system call with SO_PRIORITY is not feasible. The `netprio` module also exports another `/sys/fs/cgroup/netprio` entry, `net_prio.prioidx`. The `net_prio.prioidx` entry is a read-only file and contains a unique integer value that the kernel uses as an internal representation of this cgroup.

  `netprio` is implemented in `net/core/netprio_cgroup.c`.

  `net_cls` is implemented in `net/sched/cls_cgroup.c`.

  The network classifier cgroup provides an interface to tag network packets with a class identifier (`classid`). Creating a `net_cls` cgroups instance creates a `net_cls.classid` control file. This `net_cls.classid` value is initialized to 0. You can set up rules for this classid with `tc`, the traffic control command of `iproute2`.

  For more information, see `Documentation/cgroups/net_cls.txt`.

- `struct phy_device *phydev`

  The associated PHY device. The `phy_device` is the Layer 1 (the physical layer) device. It is defined in `include/linux/phy.h`. For many devices, PHY flow control parameters like autonegotiation, speed, or duplex can be configured via the PHY device with `ethtool` commands. See `man 8 ethtool` for more info.

- `int group`

  The group that the network device belongs to. It is initialized with INIT_NETDEV_GROUP (0) by default. The group is exported by `sysfs` via `/sys/class/net/<devName>/netdev_group`. The network device group filters are used for example in netfilter, in `net/netfilter/xt_devgroup.c`.

  Helper method:

  - `void dev_set_group(struct net_device *dev, int new_group)`: Changes the group of the specified device to be the specified group.

- `struct pm_qos_request pm_qos_req`

  Power Management Quality Of Service request object, defined in `include/linux/pm_qos.h`.

  For more details about PM QoS, see `Documentation/power/pm_qos_interface.txt`.

Next I will describe the netdev_priv() method and the alloc_netdev() macro, which are used a lot in network drivers.

The `netdev_priv(struct net_device *netdev)` method returns a pointer to the end of the `net_device`. This area is used by drivers, which define a private network interface structure in order to store private data. For example, in `drivers/net/ethernet/intel/e1000e/netdev.c`:

```
static int e1000_open(struct net_device *netdev)
{
    struct e1000_adapter *adapter = netdev_priv(netdev);
    . . .
}
```

The `netdev_priv()` method is used also for software devices, like the VLAN device. So you have:

```
static inline struct vlan_dev_priv *vlan_dev_priv(const struct net_device *dev)
{
    return netdev_priv(dev);
}
```

(`net/8021q/vlan.h`)

- The `alloc_netdev(sizeof_priv, name, setup)` macro is for allocation and initialization of a network device. It is in fact a wrapper around `alloc_netdev_mqs()`, with one Tx queue and one Rx queue. `sizeof_priv` is the size of private data to allocate space for. The `setup` method is a callback to initialize the network device. For Ethernet devices, it is usually `ether_setup()`.

  For Ethernet devices, you can use the `alloc_etherdev()` or `alloc_etherdev_mq()` macros, which eventually invoke `alloc_etherdev_mqs()`; `alloc_etherdev_mqs()` is also a wrapper around `alloc_netdev_mqs()`, with the `ether_setup()` as the setup callback method.

- Software devices usually define a setup method of their own. So, in PPP you have the `ppp_setup()` method in `drivers/net/ppp/ppp_generic.c`, and for VLAN you have `vlan_setup(struct net_device *dev)` in `net/8021q/vlan.h`.

RDMA (Remote DMA)

The following sections describe the RDMA API for the following data structures:

- RDMA device
- Protection Domain (PD)
- eXtended Reliable Connected (XRC)
- Shared Receive Queue (SRQ)
- Address Handle (AH)
- Multicast Groups
- Completion Queue (CQ)
- Queue Pair (QP)
- Memory Window (MW)
- Memory Region (MR)

RDMA Device

The following methods are related to the RDMA device.

The ib_register_client() Method

The `ib_register_client()` method registers a kernel client that wants to use the RDMA stack. The specified callbacks will be called for every RDMA device that currently exists in the system and for every new device that will be detected or removed by the system (using hot-plug). It will return 0 on success or the errno value with the reason for the failure.

```
int ib_register_client(struct ib_client *client);
```

- `client`: A structure that describes the attributes of the registration.

The ib_client Struct:

The device registration attributes are represented by `struct ib_client`:

```
struct ib_client {
        char  *name;
        void (*add)   (struct ib_device *);
        void (*remove)(struct ib_device *);

        struct list_head list;
};
```

- `name`: The name of the kernel module to be registered.
- `add`: A callback to be called for each RDMA device that exists in the system and for every new RDMA device that will be detected by the kernel.
- `remove`: A callback to be called for each RDMA device being removed by the kernel.

The ib_unregister_client() Method

The `ib_unregister_client()` method unregisters a kernel module that wants to stop using the RDMA stack.

```
void ib_unregister_client(struct ib_client *client);
```

- `device`: A structure that describes the attributes of the unregistration.
- `client`: Should be the same object that was used when `ib_register_client()` was called.

The ib_get_client_data() Method

The `ib_get_client_data()` method returns the client context which was associated with the RDMA device using the `ib_set_client_data()` method.

```
void *ib_get_client_data(struct ib_device *device, struct ib_client *client);
```

- `device`: The RDMA device to get the client context from.
- `client`: The object that describes the attributes of the registration/unregistration.

The ib_set_client_data() Method

The `ib_set_client_data()` method sets a client context to be associated with the RDMA device.

```
void  ib_set_client_data(struct ib_device *device, struct ib_client *client,
             void *data);
```

- `device`: The RDMA device to set the client context with.
- `client`: The object that describes the attributes of the registration/unregistration.
- `data`: The client context to associate.

The INIT_IB_EVENT_HANDLER macro

The INIT_IB_EVENT_HANDLER macro initializes an event handler for the asynchronous events that may occur to the RDMA device. This macro should be used before calling the `ib_register_event_handler()` method:

```
#define INIT_IB_EVENT_HANDLER(_ptr, _device, _handler)        \
    do {                            \
        (_ptr)->device  = _device;          \
        (_ptr)->handler = _handler;          \
        INIT_LIST_HEAD(&(_ptr)->list);          \
    } while (0)
```

- `_ptr`: A pointer to the event handler that will be provided to the `ib_register_event_handler()` method.
- `_device`: The RDMA device context; upon its events the callback will be called.
- `_handler`: The callback that will be called with every asynchronous event.

The ib_register_event_handler() Method

The `ib_register_event_handler()` method registers an RDMA event to be called with every handler asynchronous event. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_register_event_handler  (struct ib_event_handler *event_handler);
```

- `event_handler`: The event handler that was initialized with the macro INIT_IB_EVENT_HANDLER. This callback may occur in interrupt context.

The ib_event_handler struct:

The RDMA event handler is represented by `struct ib_event_handler`:

```
struct ib_event_handler {
    struct ib_device *device;
    void            (*handler)(struct ib_event_handler *, struct ib_event *);
    struct list_head  list;
};
```

The ib_event Struct

The event callback is being called with the new event that happens to the RDMA device. This event is represented by `struct ib_event`.

```
struct ib_event {
    struct ib_device    *device;
    union {
        struct ib_cq    *cq;
        struct ib_qp    *qp;
        struct ib_srq    *srq;
        u8        port_num;
    } element;
    enum ib_event_type    event;
};
```

- `device`: The RDMA device to which the asynchronous event occurred.
- `element.cq`: If this is a CQ event, the CQ on which the asynchronous event occurred.
- `element.qp`: If this is a QP event, the QP on which the asynchronous event occurred.
- `element.srq`: If this is an SRQ event, the SRQ on which the asynchronous event occurred.
- `element.port_num`: If this is a port event, the port number on which the asynchronous event occurred.
- `event`: The type of the asynchronous event that was occurred. It can be:
    - IB_EVENT_CQ_ERR: CQ event. An error occurred to the CQ and no more Work Completions will be generated to it.
    - IB_EVENT_QP_FATAL: QP event. An error occurred to the QP that prevents it from reporting an error through a Work Completion.
    - IB_EVENT_QP_REQ_ERR: QP event. An incoming RDMA request caused a transport error violation in the targeted QP.
    - IB_EVENT_QP_ACCESS_ERR: QP event. An incoming RDMA request caused a requested error violation in the targeted QP.
    - IB_EVENT_COMM_EST: QP event. A communication established event occurred. An incoming message was received by a QP when it was in the RTR state.
    - IB_EVENT_SQ_DRAINED: QP event. Send Queue drain event. The QP's Send Queue was drained.
    - IB_EVENT_PATH_MIG: QP event. Path migration was completed successfully and the primary was changed.
    - IB_EVENT_PATH_MIG_ERR: QP event. There was an error when trying to perform path migration.
    - IB_EVENT_DEVICE_FATAL: Device event. There was an error with the RDMA device.
    - IB_EVENT_PORT_ACTIVE: Port event. The port state has become active.
    - IB_EVENT_PORT_ERR: Port event. The port state was active and it is no longer active.
    - IB_EVENT_LID_CHANGE: Port event. The LID of the port was changed.
    - IB_EVENT_PKEY_CHANGE: Port event. A P_Key entry was changed in the port's P_Key table.
    - IB_EVENT_SM_CHANGE: Port event. The Subnet Manager that manages this port was change.
    - IB_EVENT_SRQ_ERR: SRQ event. An error occurred to the SRQ.
    - IB_EVENT_SRQ_LIMIT_REACHED: SRQ event/SRQ limit event. The number of Receive Requests in the SRQ dropped below the requested watermark.

- IB_EVENT_QP_LAST_WQE_REACHED: QP event. Last Receive Request reached from the SRQ, and it won't consume any more Receive Requests from it.
- IB_EVENT_CLIENT_REREGISTER: Port event. The client should reregister to all services from the Subnet Administrator.
- IB_EVENT_GID_CHANGE: Port event. A GID entry was changed in the port's GID table.

The ib_unregister_event_handler() Method

The `ib_unregister_event_handler()` method unregisters an RDMA event handler. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_unregister_event_handler(struct ib_event_handler *event_handler);
```

- `event_handler`: The event handler to be unregistered. It should be the same object that was registered with `ib_register_event_handler()`.

The ib_query_device() Method

The `ib_query_device()` method queries the RDMA device for its attributes. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_query_device(struct ib_device *device,
        struct ib_device_attr *device_attr);
```

- device: The RDMA device to be queried.
- `device_attr`: Pointer to a structure of an RDMA device attributes that will be filled.

The ib_device_attr struct:

The RDMA device attributes are represented by `struct ib_device_attr`:

```
struct ib_device_attr {
    u64             fw_ver;
    __be64          sys_image_guid;
    u64             max_mr_size;
    u64             page_size_cap;
    u32             vendor_id;
    u32             vendor_part_id;
    u32             hw_ver;
    int             max_qp;
    int             max_qp_wr;
    int             device_cap_flags;
    int             max_sge;
    int             max_sge_rd;
    int             max_cq;
    int             max_cqe;
    int             max_mr;
    int             max_pd;
    int             max_qp_rd_atom;
    int             max_ee_rd_atom;
    int             max_res_rd_atom;
    int             max_qp_init_rd_atom;
    int             max_ee_init_rd_atom;
    enum ib_atomic_cap    atomic_cap;
    enum ib_atomic_cap    masked_atomic_cap;
    int             max_ee;
    int             max_rdd;
    int             max_mw;
    int             max_raw_ipv6_qp;
    int             max_raw_ethy_qp;
    int             max_mcast_grp;
    int             max_mcast_qp_attach;
    int             max_total_mcast_qp_attach;
    int             max_ah;
    int             max_fmr;
    int             max_map_per_fmr;
    int             max_srq;
    int             max_srq_wr;
    int             max_srq_sge;
    unsigned int    max_fast_reg_page_list_len;
    u16             max_pkeys;
    u8              local_ca_ack_delay;
};
```

- fw_ver: A number which represents the FW version of the RDMA device. It can be evaluated as ZZZZYYXX: Zs are the major number, Ys are the minor number, and Xs are the build number.
- sys_image_guid: The system image GUID: Has a unique value for each system.
- max_mr_size: The maximum supported MR size.
- page_size_cap: Bitwise OR for all of supported memory page shifts.
- vendor_id: The IEEE vendor ID.
- vendor_part_id: Device's part ID, as supplied by the vendor.
- hw_ver: Device's HW version, as supplied by the vendor.
- max_qp: Maximum supported number of QPs.
- max_qp_wr: Maximum supported number of Work Requests in each non-RD QP.
- device_cap_flags: Supported capabilities of the RDMA device. It is a bitwise OR of the masks:
    - IB_DEVICE_RESIZE_MAX_WR: The RDMA device supports resize of the number of Work Requests in a QP.
    - IB_DEVICE_BAD_PKEY_CNTR: The RDMA device supports the ability to count the number of bad P_Keys.
    - IB_DEVICE_BAD_QKEY_CNTR: The RDMA device supports the ability to count the number of bad Q_Keys.

- IB_DEVICE_RAW_MULTI: The RDMA device supports raw packet multicast.
- IB_DEVICE_AUTO_PATH_MIG: The RDMA device supports Automatic Path Migration.
- IB_DEVICE_CHANGE_PHY_PORT: The RDMA device supports changing the QP's primary Port number.
- IB_DEVICE_UD_AV_PORT_ENFORCE: The RDMA device supports enforcements of the port number of UD QP and Address Handle.
- IB_DEVICE_CURR_QP_STATE_MOD: The RDMA device supports the current QP modifier when calling `ib_modify_qp()`.
- IB_DEVICE_SHUTDOWN_PORT: The RDMA device supports port shutdown.
- IB_DEVICE_INIT_TYPE: The RDMA device supports setting InitType and InitTypeReply.
- IB_DEVICE_PORT_ACTIVE_EVENT: The RDMA device supports the generation of the port active asynchronous event.
- IB_DEVICE_SYS_IMAGE_GUID: The RDMA device supports system image GUID.
- IB_DEVICE_RC_RNR_NAK_GEN: The RDMA device supports RNR-NAK generation for RC QPs.
- IB_DEVICE_SRQ_RESIZE: The RDMA device supports resize of a SRQ.
- IB_DEVICE_N_NOTIFY_CQ: The RDMA device supports notification when N Work Completions exists in the CQ.
- IB_DEVICE_LOCAL_DMA_LKEY: The RDMA device supports Zero Stag (in iWARP) and reserved LKey (in InfiniBand).
- IB_DEVICE_RESERVED: Reserved bit.
- IB_DEVICE_MEM_WINDOW: The RDMA device supports Memory Windows.
- IB_DEVICE_UD_IP_CSUM: The RDMA device supports insertion of UDP and TCP checksum on outgoing UD IPoIB messages and can verify the validity of those checksum for incoming messages.
- IB_DEVICE_UD_TSO: The RDMA device supports TCP Segmentation Offload.
- IB_DEVICE_XRC: The RDMA device supports the eXtended Reliable Connected transport.
- IB_DEVICE_MEM_MGT_EXTENSIONS: The RDMA device supports memory management extensions support.
- IB_DEVICE_BLOCK_MULTICAST_LOOPBACK: The RDMA device supports blocking multicast loopback.
- IB_DEVICE_MEM_WINDOW_TYPE_2A: The RDMA device supports Memory Windows type 2A: association with a QP number.
- IB_DEVICE_MEM_WINDOW_TYPE_2B: The RDMA device supports Memory Windows type 2B: association with a QP number and a PD.

- max_sge: Maximum supported number of scatter/gather elements per Work Request in a non-RD QP.
- max_sge_rd: Maximum supported number of scatter/gather elements per Work Request in an RD QP.
- max_cq: Maximum supported number of CQs.
- max_cqe: Maximum supported number of entries in each CQ.
- max_mr: Maximum supported number of MRs.
- max_pd: Maximum supported number of PDs.
- max_qp_rd_atom: Maximum number of RDMA Read and Atomic operations that can be sent to a QP as the target of the operation.
- max_ee_rd_atom: Maximum number of RDMA Read and Atomic operations that can be sent to an EE context as the target of the operation.
- max_res_rd_atom: Maximum number of for incoming RDMA Read and Atomic operations that can be sent to this RDMA device as the target of the operation.
- max_qp_init_rd_atom: Maximum number of RDMA Read and Atomic operations that can be sent from a QP as the initiator of the operation.
- max_ee_init_rd_atom: Maximum number of RDMA Read and Atomic operations that can be sent from an EE context as the initiator of the operation.
- atomic_cap: Ability of the device to support atomic operations. Can be:
    - IB_ATOMIC_NONE: The RDMA device doesn't guarantee any atomicity at all.
    - IB_ATOMIC_HCA: The RDMA device guarantees atomicity between QPs in the same device.
    - IB_ATOMIC_GLOB: The RDMA device guarantees atomicity between this device and any other component.
- masked_atomic_cap: The ability of the device to support masked atomic operations. Possible values as described in atomic_cap earlier.
- max_ee: Maximum supported number of EE contexts.
- max_rdd: Maximum supported number of RDDs.
- max_mw: Maximum supported number of MWs.
- max_raw_ipv6_qp: Maximum supported number of Raw IPv6 Datagram QPs.
- max_raw_ethy_qp: Maximum supported number of Raw Ethertype Datagram QPs.
- max_mcast_grp: Maximum supported number of multicast groups.
- max_mcast_qp_attach: Maximum supported number of QPs that can be attached to each multicast group.
- max_total_mcast_qp_attach: Maximum number of total QPs that can be attached to any multicast group.
- max_ah: Maximum supported number of AHs.
- max_fmr: Maximum supported number of FMRs.

- `max_map_per_fmr`: Maximum supported number of map operations which are allowed per FMR.
- `max_srq`: Maximum supported number of SRQs.
- `max_srq_wr`: Maximum supported number of Work Requests in each SRQ.
- `max_srq_sge`: Maximum supported number of scatter/gather elements per Work Request in an SRQ.
- `max_fast_reg_page_list_len`: Maximum number of page list that can be used when registering an FMR using a Work Request.
- `max_pkeys`: Maximum supported number of P_Keys.
- `local_ca_ack_delay`: Local CA ack delay. This value specifies the maximum expected time interval between the local device receiving a message and transmitting the associated ACK or NAK.

The ib_query_port() Method

The `ib_query_port()` method queries the RDMA device port's attributes. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_query_port(struct ib_device *device,
         u8 port_num, struct ib_port_attr *port_attr);
```

- `device`: The RDMA device to be queried.
- `port_num`: The port number to be queried.
- `port_attr`: A pointer to a structure of an RDMA port attributes which will be filled.

The ib_port_attr Struct

The RDMA port attributes are represented by `struct ib_port_attr`:

```
struct ib_port_attr {
    enum ib_port_state    state;
    enum ib_mtu   max_mtu;
    enum ib_mtu   active_mtu;
    int           gid_tbl_len;
    u32           port_cap_flags;
    u32           max_msg_sz;
    u32           bad_pkey_cntr;
    u32           qkey_viol_cntr;
    u16           pkey_tbl_len;
    u16           lid;
    u16           sm_lid;
    u8            lmc;
    u8            max_vl_num;
    u8            sm_sl;
    u8            subnet_timeout;
    u8              init_type_reply;
    u8              active_width;
    u8              active_speed;
    u8              phys_state;
};
```

- `state`: The logical port state. Can be:
  - IB_PORT_NOP: Reserved value.
  - IB_PORT_DOWN: Logical link is down.
  - IB_PORT_INIT: Logical link is initialized. The physical link is up but the Subnet Manager hasn't started to configure the port.
  - IB_PORT_ARMED: Logical link is armed. The physical link is up but the Subnet Manager started, and did not yet complete, configuring the port.
  - IB_PORT_ACTIVE: Logical link is active.
  - IB_PORT_ACTIVE_DEFER: Logical link is active but the physical link is down. The link tries to recover from this state.
- `max_mtu`: The maximum MTU supported by this port. Can be:
  - IB_MTU_256: 256 bytes.
  - IB_MTU_512: 512 bytes.
  - IB_MTU_1024: 1,024 bytes.
  - IB_MTU_2048: 2,048 bytes.
  - IB_MTU_4096: 4,096 bytes.
- `active_mtu`: The actual MTU that this port is configured with. Can be as `max_mtu`, mentioned earlier.
- `gid_tbl_len`: The number of entries in the port's GID table.
- `port_cap_flags`: The port supported capabilities. It is a bitwise OR of the masks:
  - IB_PORT_SM: An indication that the SM that manages the subnet is sending packets from this port.
  - IB_PORT_NOTICE_SUP: An indication that this port supports notices.
  - IB_PORT_TRAP_SUP: An indication that this port supports traps.
  - IB_PORT_OPT_IPD_SUP: An indication that this port supports Inter Packet Delay optional values.
  - IB_PORT_AUTO_MIGR_SUP: An indication that this port supports Automatic Path Migration.
  - IB_PORT_SL_MAP_SUP: An indication that this port supports SL 2 VL mapping table.
  - IB_PORT_MKEY_NVRAM: An indication that this port supports saving the M_Key attributes in Non Volatile RAM.

    - IB_PORT_PKEY_NVRAM: An indication that this port supports saving the P_Key table in Non Volatile RAM.
    - IB_PORT_LED_INFO_SUP: An indication that this port supports turning on and off the LED using management packets.
    - IB_PORT_SM_DISABLED: An indication that there is an SM which isn't active in this port.
    - IB_PORT_SYS_IMAGE_GUID_SUP: An indication that the port supports system image GUID.
    - IB_PORT_PKEY_SW_EXT_PORT_TRAP_SUP: An indication that the SMA on the switch management port will monitor P_Key mismatches on each switch external port.
    - IB_PORT_EXTENDED_SPEEDS_SUP: An indication that the port supports extended speeds (FDR and EDR).
    - IB_PORT_CM_SUP: An indication that this port supports CM.
    - IB_PORT_SNMP_TUNNEL_SUP: An indication that an SNMP tunneling agent is listening on this port.
    - IB_PORT_REINIT_SUP: An indication that this port supports reinitialization of the node.
    - IB_PORT_DEVICE_MGMT_SUP: An indication that this port supports device management.
    - IB_PORT_VENDOR_CLASS_SUP: An indication that a vendor-specific agent is listening on this port.
    - IB_PORT_DR_NOTICE_SUP: An indication that this port supports Direct Route notices.
    - IB_PORT_CAP_MASK_NOTICE_SUP: An indication that this port supports sending a notice if the port's `port_cap_flags` is changed.
    - IB_PORT_BOOT_MGMT_SUP: An indication that a boot manager agent is listening on this port.
    - IB_PORT_LINK_LATENCY_SUP: An indication that this port supports link round trip latency measurement.
    - IB_PORT_CLIENT_REG_SUP: An indication that this port is capable of generating the IB_EVENT_CLIENT_REREGISTER asynchronous event.
- `max_msg_sz:` The maximum supported message size by this port.
- `bad_pkey_cntr:` A counter for the number of bad P_Key from messages that this port received.
- `qkey_viol_cntr:` A counter for the number of Q_Key violations from messages that this port received.
- `pkey_tbl_len`: The number of entries in the port's P_Key table.
- `lid`: The port's Local Identifier (LID), as assigned by the SM.
- `sm_lid`: The LID of the SM.
- `lmc`: LID mask of this port.

- `max_vl_num`: Maximum number of Virtual Lanes supported by this port. Can be:
    - 1: 1 VL is supported: VL0
    - 2: 2 VLs are supported: VL0–VL1
    - 3: 4 VLs are supported: VL0–VL3
    - 4: 8 VLs are supported: VL0–VL7
    - 5: 15 VLs are supported: VL0–VL14
- `sm_sl`: The SL to be used when sending messages to the SM.
- `subnet_timeout`: The maximum expected subnet propagation delay. This duration of time calculation is 4.094*2^subnet_timeout.
- `init_type_reply`: The value that the SM configures before moving the port state to IB_PORT_ARMED or IB_PORT_ACTIVE to specify the type of the initialization performed.
- `active_width`: The port's active width. Can be:
    - IB_WIDTH_1X: Multiple of 1.
    - IB_WIDTH_4X: Multiple of 4.
    - IB_WIDTH_8X: Multiple of 8.
    - IB_WIDTH_12X: Multiple of 12.
- `active_speed`: The port's active speed. Can be:
    - IB_SPEED_SDR: Single Data Rate (SDR): 2.5 Gb/sec, 8/10 bit encoding.
    - IB_SPEED_DDR: Double Data Rate (DDR): 5 Gb/sec, 8/10 bit encoding.
    - IB_SPEED_QDR: Quad Data Rate (DDR): 10 Gb/sec, 8/10 bit encoding.
    - IB_SPEED_FDR10: Fourteen10 Data Rate (FDR10): 10.3125 Gb/sec, 64/66 bit encoding.
    - IB_SPEED_FDR: Fourteen Data Rate (FDR): 14.0625 Gb/sec, 64/66 bit encoding.
    - IB_SPEED_EDR: Enhanced Data Rate (EDR): 25.78125 Gb/sec.
- `phys_state`: The physical port state. There isn't any enumeration for this value.

The rdma_port_get_link_layer() Method

The `rdma_port_get_link_layer()` method returns the link layer of the RDMA device port. It will return the following values:

- IB_LINK_LAYER_UNSPECIFIED: Unspecified value, usually legacy value that indicates that this is an InfiniBand link layer.
- IB_LINK_LAYER_INFINIBAND: Link layer is InfiniBand.
- IB_LINK_LAYER_ETHERNET: Link layer is Ethernet. This indicates that the port supports RDMA Over Converged Ethernet (RoCE).

```
enum rdma_link_layer rdma_port_get_link_layer(struct ib_device *device, u8 port_num);
```

- device: The RDMA device to be queried.
- port_num: The port number to be queried.

The ib_query_gid() Method

The ib_query_gid() method queries the RDMA device port's GID table. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_query_gid(struct ib_device *device, u8 port_num, int index, union ib_gid *gid);
```

- device: The RDMA device to be queried.
- port_num: The port number to be queried.
- index: The index in the GID table to be queried.
- gid: A pointer to the GID union to be filled.

The ib_query_pkey() Method

The ib_query_pkey() method queries the RDMA device port's P_Key table. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_query_pkey(struct ib_device *device,
        u8 port_num, u16 index, u16 *pkey);
```

- device: The RDMA device to be queried.
- port_num: The port number to be queried.
- index: The index in the P_Key table to be queried.
- pkey: A pointer to the P_Key to be filled.

The ib_modify_device() Method

The ib_modify_device() method modifies the RDMA device attributes. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_modify_device(struct ib_device *device,
            int device_modify_mask,
            struct ib_device_modify *device_modify);
```

- device: The RDMA device to be modified.
- device_modify_mask: The device attributes to be changed. It is a bitwise OR of the masks:
    - IB_DEVICE_MODIFY_SYS_IMAGE_GUID: Modifies the system image GUID.
    - IB_DEVICE_MODIFY_NODE_DESC: Modifies the node description.
- device_modify: The RDMA attributes to be modified, as described immediately.

The ib_device_modify Struct

The RDMA device attributes are represented by `struct ib_device_modify`:

```
struct ib_device_modify {
    u64     sys_image_guid;
    char     node_desc[64];
};
```

- `sys_image_guid`: A 64-bit value of the system image GUID.
- `node_desc`: A NULL terminated string that describes the node description.

The ib_modify_port() Method

The `ib_modify_port()` method modifies the RDMA device port's attributes. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_modify_port(struct ib_device *device,
           u8 port_num, int port_modify_mask,
           struct ib_port_modify *port_modify);
```

- `device`: The RDMA device to be modified.
- `port_num`: The port number to be modified.
- `port_modify_mask`: The port's attributes to be changed. It is a bitwise OR of the masks:
    - IB_PORT_SHUTDOWN: Moves the port state to IB_PORT_DOWN.
    - IB_PORT_INIT_TYPE: Sets the port InitType value.
    - IB_PORT_RESET_QKEY_CNTR: Resets the port's Q_Key violation counter.
- `port_modify`: The port attributes to be modified, as described in the next section.

The ib_port_modify struct:

The RDMA device attributes are represented by `struct ib_port_modify`:

```
struct ib_port_modify {
    u32     set_port_cap_mask;
    u32     clr_port_cap_mask;
    u8     init_type;
};
```

- `set_port_cap_mask`: The port capabilities bits to be set.
- `clr_port_cap_mask`: The port capabilities bits to be cleared.
- `init_type`: The InitType value to be set.

The ib_find_gid() Method

The `ib_find_gid()` method finds the port number and the index where a specific GID value exists in the GID table. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_find_gid(struct ib_device *device, union ib_gid *gid,
        u8 *port_num, u16 *index);
```

- `device`: The RDMA device to be queried.
- `gid`: A pointer of the GID to search for.
- `port_num`: Will be filled with the port number that this GID exists in.
- `index`: Will be filled with the index in the GID table that this GID exists in.

The ib_find_pkey() Method

The `ib_find_pkey()` method finds the index where a specific P_Key value exists in the P_Key table in a specific port number. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_find_pkey(struct ib_device *device,
         u8 port_num, u16 pkey, u16 *index);
```

- `device`: The RDMA device to be queried.
- `port_num`: The port number to search the P_Key in.
- `pkey`: The P_Key value to search for.
- `index`: The index in the P_Key table that this P_Key exists in.

The rdma_node_get_transport() Method

The `rdma_node_get_transport()` method returns the RDMA transport type of a specific node type. The available transport types can be:

- RDMA_TRANSPORT_IB: Transport is InfiniBand.
- RDMA_TRANSPORT_IWARP: Transport is iWARP.

The rdma_node_get_transport() Method

```
enum rdma_transport_type
rdma_node_get_transport(enum rdma_node_type node_type) __attribute_const__;
```

- `node_type`: The node type. Can be:RDMA_NODE_IB_CA: Node type is an InfiniBand Channel Adapter.
- RDMA_NODE_IB_SWITCH: Node type is an InfiniBand Switch.
- RDMA_NODE_IB_ROUTER: Node type is an InfiniBand Router.
- RDMA_NODE_RNIC: Node type is an RDMA NIC.

The ib_mtu_to_int() Method

The `ib_mtu_to_int()` method returns the number of bytes, as an integer, for MTU enumerations. It will return a positive value on success or –1 on a failure.

```
static inline int ib_mtu_enum_to_int(enum ib_mtu mtu);
```

- `mtu`: Can be an MTU enumeration, as described earlier.

The ib_width_enum_to_int() Method

The `ib_width_enum_to_int()` method returns the number of width multiple, as an integer, for an IB port enumerations. It will return a positive value on success or –1 on a failure.

```
static inline int ib_width_enum_to_int(enum ib_port_width width);
```

- `width`: Can be a port width enumeration, as described earlier.

The ib_rate_to_mult() Method

The `ib_rate_to_mult()` method returns the number of multiple of the base rate of 2.5 Gbit/sec, as an integer, for an IB rate enumerations. It will return a positive value on success or –1 on a failure.

```
int ib_rate_to_mult(enum ib_rate rate) __attribute_const__;
```

- rate: The rate enumeration to be converted. Can be:
  - IB_RATE_PORT_CURRENT: Current port's rate.
  - IB_RATE_2_5_GBPS: Rate of 2.5 Gbit/sec.
  - IB_RATE_5_GBPS: Rate of 5 Gbit/sec.
  - IB_RATE_10_GBPS: Rate of 10 Gbit/sec.
  - IB_RATE_20_GBPS: Rate of 20 Gbit/sec.
  - IB_RATE_30_GBPS: Rate of 30 Gbit/sec.
  - IB_RATE_40_GBPS: Rate of 40 Gbit/sec.
  - IB_RATE_60_GBPS: Rate of 60 Gbit/sec.
  - IB_RATE_80_GBPS: Rate of 80 Gbit/sec.
  - IB_RATE_120_GBPS: Rate of 120 Gbit/sec.
  - IB_RATE_14_GBPS: Rate of 14 Gbit/sec.
  - IB_RATE_56_GBPS: Rate of 56 Gbit/sec.
  - IB_RATE_112_GBPS: Rate of 112 Gbit/sec.
  - IB_RATE_168_GBPS: Rate of 168 Gbit/sec.
  - IB_RATE_25_GBPS: Rate of 25 Gbit/sec.

- IB_RATE_100_GBPS: Rate of 100 Gbit/sec.
- IB_RATE_200_GBPS: Rate of 200 Gbit/sec.
- IB_RATE_300_GBPS: Rate of 300 Gbit/sec.

The ib_rate_to_mbps() Method

The `ib_rate_to_mbps()` method returns the number of Mbit/sec, as an integer, for an IB rate enumerations. It will return a positive value on success or –1 on a failure.

```
int ib_rate_to_mbps(enum ib_rate rate) __attribute_const__;
```

- `rate`: The rate enumeration to be converted, as described earlier.

The ib_rate_to_mbps() Method

The `ib_rate_to_mbps()` method returns the IB rate enumerations for a multiple of the base rate of 2.5 Gbit/sec. It will return a positive value on success or –1 on a failure.

```
enum ib_rate mult_to_ib_rate(int mult) __attribute_const__;
```

- `mult`: The rate multiple to be converted, as described earlier.

Protection Domain (PD)

PD is an RDMA resource that associates QPs and SRQs with MRs and AHs with QPs. One can look at PD as a color, for example: red MR can work with a red QP, and red AH can work with a red QP. Working with green AH with a red QP will result in an error.

The ib_alloc_pd() Method

The `ib_alloc_pd()` method allocates a PD. It will return a pointer to the newly allocated PD on success or an ERR_PTR() which specifies the reason for the failure.

```
struct ib_pd *ib_alloc_pd(struct ib_device *device);
```

- `device`: The RDMA device that the PD will be associated with.

The ib_dealloc_pd() Method

The `ib_dealloc_pd()` method deallocates a PD. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_dealloc_pd(struct ib_pd *pd);
```

- `pd`: The PD to be deallocated.

eXtended Reliable Connected (XRC)

XRC is an IB transport extension that provides better scalability, in the sender side, for Reliable Connected QPs than the original Reliable Transport can provide. Using XRC will decrease the number of QPs between two specific cores: when using RC QPs, for each core, in each machine, there is a QP. When using XRC, there will be one XRC QP in each host. When sending a message, the sender needs to specify the remote SRQ number that will receive the message.

The ib_alloc_xrcd() Method

The `ib_alloc_xrcd()` method allocates an XRC domain. It will return a pointer to the newly created XRC domain on success or an ERR_PTR() which specifies the reason for the failure.

```
struct ib_xrcd *ib_alloc_xrcd(struct ib_device *device);
```

- `device`: The RDMA device that this XRC domain will be allocated on.

The ib_dealloc_xrcd_cq() Method

The `ib_dealloc_xrcd_cq()` method deallocates an XRC domain. It will return 0 on success or the errno value with the reason for the failure:

```
int ib_dealloc_xrcd(struct ib_xrcd *xrcd);
```

- `xrcd`: The XRC domain to be deallocated.

Shared Receive Queue (SRQ)

SRQ is a resource that helps RDMA to be more scalable. Instead of managing the Receive Requests in the Receive Queues of many QPs, it is possible to manage them in a single Receive Queue, which all of them share. This will eliminate starvation in RC QPs or packet drops in unreliable transport types and will help to reduce the total posted Receive Requests, thus reducing the consumed memory. Furthermore, unlike a QP, an SRQ can have a watermark to allow a notification if the number of RRs in the SRQ dropped below a specify value.

The ib_srq_attr Struct

The SRQ attributes are represented by `struct ib_srq_attr`:

```
struct ib_srq_attr {
    u32    max_wr;
    u32    max_sge;
    u32    srq_limit;
};
```

- `max_wr`: The maximum number of outstanding RRs that this SRQ can hold.
- `max_sge`: The maximum number of scatter/gather elements that each RR in the SRQ can hold.
- `srq_limit`: The watermark limit that creates an asynchronous event if the number of RRs in the SRQ dropped below this value.

The ib_create_srq() Method

The ib_create_srq() method creates an SRQ. It will return a pointer to the newly created SRQ on success or an ERR_PTR() which specifies the reason for the failure:

```
struct ib_srq *ib_create_srq(struct ib_pd *pd, struct ib_srq_init_attr *srq_init_attr);
```

- pd: The PD that this SRQ is being associated with.
- srq_init_attr: The attributes that this SRQ will be created with.

The ib_srq_init_attr Struct

The created SRQ attributes are represented by struct ib_srq_init_attr:

```
struct ib_srq_init_attr {
    void              (*event_handler)(struct ib_event *, void *);
    void               *srq_context;
    struct ib_srq_attr    attr;
    enum ib_srq_type    srq_type;

    union {
        struct {
            struct ib_xrcd *xrcd;
            struct ib_cq   *cq;
        } xrc;
    } ext;
};
```

- event_handler: A pointer to a callback that will be called in case of an affiliated asynchronous event to the SRQ.
- srq_context: User-defined context that can be associated with the SRQ.
- attr: The SRQ attributes, as described earlier.
- srq_type: The type of the SRQ. Can be:
  - IB_SRQT_BASIC: For regular SRQ.
  - IB_SRQT_XRC: For XRC SRQ.
- ext: If srq_type is IB_SRQT_XRC, specifies the XRC domain or the CQ that this SRQ is associated with.

The ib_modify_srq() Method

The ib_modify_srq() method modifies the attributes of the SRQ. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_modify_srq(struct ib_srq *srq, struct ib_srq_attr *srq_attr, enum ib_srq_attr_mask srq_attr_mask);
```

- `srq`: The SRQ to be modified.
- `srq_attr`: The SRQ attributes, as described earlier.
- `srq_attr_mask`: The SRQ attributes to be changed. It is a bitwise OR of the masks:
    - IB_SRQ_MAX_WR: Modify the number of RRs in the SRQ (that is, resize the SRQ). This can be done only if the device supports SRQ resize—that is, the IB_DEVICE_SRQ_RESIZE is set in the device flags.
    - IB_SRQ_LIMIT: Set the value of the SRQ watermark limit.

The ib_query_srq() Method

The `ib_query_srq()` method queries for the current SRQ attributes. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_query_srq(struct ib_srq *srq, struct ib_srq_attr *srq_attr);
```

- `srq`: The SRQ to be queried.
- `srq_attr`: The SRQ attributes, as described earlier.

The ib_destory_srq() Method

The `ib_destory_srq()` method destroys an SRQ. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_destroy_srq(struct ib_srq *srq);
```

- `srq`: The SRQ to be destroyed.

The ib_post_srq_recv() Method

The `ib_post_srq_recv()` method takes a linked list of Receive Requests and adds them to the SRQ for future processing. Every Receive Request is considered outstanding until a Work Completion is generated after its processing. It will return 0 on success or the errno value with the reason for the failure.

```
static inline int ib_post_srq_recv(struct ib_srq *srq, struct ib_recv_wr *recv_wr,
struct ib_recv_wr **bad_recv_wr);
```

- `srq`: The SRQ that the Receive Requests will be posted to.
- `recv_wr`: A linked list of Receive Request to be posted.
- `bad_recv_wr`: If there was an error with the handling of the Receive Requests, this pointer will be filled with the address of the Receive Request that caused this error.

The ib_recv_wr Struct

The Receive Request is represented by `struct ib_recv_wr`:

```
struct ib_recv_wr {
    struct ib_recv_wr          *next;
    u64            wr_id;
    struct ib_sge       *sg_list;
    int            num_sge;
};
```

- `next`: A pointer to the next Receive Request in the list or NULL, if this is the last Receive Request.
- `wr_id`: A 64-bit value that is associated with this Receive Request and will be available in the corresponding Work Completion.
- `sg_list`: The array of the scatter/gather elements, as described in the next section.
- `num_sge`: The number of entries in `sg_list`. The value zero means that the message size that can be saved has zero bytes.

The ib_sge Struct

The scatter/gather element is represented by `struct ib_sge`:

```
struct ib_sge {
    u64    addr;
    u32    length;
    u32    lkey;
};
```

- `addr`: The address of the buffer to access.
- `length`: The length of the address to access.
- `lkey`: The Local Key of the Memory Region that this buffer was registered with.

Address Handle (AH)

AH is an RDMA resource that describes the path from the local port to the remote port of the destination. It is being used for a UD QP.

The ib_ah_attr Struct

The AH attributes are represented by `struct ib_ah_attr`:

```
struct ib_ah_attr {
    struct ib_global_route    grh;
    u16               dlid;
    u8               sl;
    u8               src_path_bits;
    u8               static_rate;
    u8               ah_flags;
    u8               port_num;
};
```

- `grh`: The Global Routing Header attributes that are used for sending messages to another subnet or to a multicast group in the local or remote subnet.
- `dlid`: The destination LID.
- `sl`: The Service Level that this message will use.
- `src_path_bits`: The used source path bits. Relevant if LMC is used in this port.
- `static_rate`: The level of delay that should be done between sending the messages. It is used when sending a message to a remote node that supports a slower message rate than the local node.
- `ah_flags`: The AH flags. It is a bitwise OR of the masks:
    - IB_AH_GRH: GRH is used in this AH.
- `port_num`: The local port number that messages will be sent from.

The ib_create_ah() Method

The `ib_create_ah()` method creates an AH. It will return a pointer to the newly created AH on success or an ERR_PTR() which specifies the reason for the failure.

```
struct ib_ah *ib_create_ah(struct ib_pd *pd, struct ib_ah_attr *ah_attr);
```

- `pd`: The PD that this AH is being associated with.
- `ah_attr`: The attributes that this AH will be created with.

The ib_init_ah_from_wc() Method

The `ib_init_ah_from_wc()` method initializes an AH attribute structure from a Work Completion and a GRH structure. This is being done in order to return a message back for an incoming message of an UD QP. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_init_ah_from_wc(struct ib_device *device, u8 port_num, struct ib_wc *wc,
        struct ib_grh *grh, struct ib_ah_attr *ah_attr);
```

- `device`: The RDMA device that the Work Completion came from and the AH to be created on.
- `port_num`: The port number that the Work Completion came from and the AH will be associated with.
- `wc`: The Work Completion of the incoming message.
- `grh`: The GRH buffer of the incoming message.
- `ah_attr`: The attributes of this AH to be filled.

The ib_create_ah_from_wc() Method

The `ib_create_ah_from_wc()` method creates an AH from a Work Completion and a GRH structure. This is done in order to return a message back for an incoming message of a UD QP. It will return a pointer to the newly created AH on success or an ERR_PTR() which specifies the reason for the failure.

```
struct ib_ah *ib_create_ah_from_wc(struct ib_pd *pd, struct ib_wc *wc, struct ib_grh *grh, u8 port_num);
```

- `pd`: The PD that this AH is being associated with.
- `wc`: The Work Completion of the incoming message.
- `grh`: The GRH buffer of the incoming message.
- `port_num`: The port number that the Work Completion came from and the AH will be associated with.

The ib_modify_ah() Method

The `ib_modify_ah()` method modifies the attributes of the AH. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_modify_ah(struct ib_ah *ah, struct ib_ah_attr *ah_attr);
```

- `ah`: The AH to be modified.
- `ah_attr`: The AH attributes, as described earlier.

The ib_query_ah() Method

The `ib_query_ah()` method queries for the current AH attributes. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_query_ah(struct ib_ah *ah, struct ib_ah_attr *ah_attr);
```

- `ah`: The AH to be queried
- `ah_attr`: The AH attributes, as described earlier.

The ib_destory_ah() Method

The `ib_destory_ah()` method destroys an AH. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_destroy_ah(struct ib_ah *ah);
```

- `ah`: The AH to be destroyed.

Multicast Groups

Multicast groups are means to send a message from one UD QP to many UD QPs. Every UD QP that wants to get this message needs to be attached to a multicast group.

The ib_attach_mcast() Method

The `ib_attach_mcast()` method attaches a UD QP to a multicast group within an RDMA device. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_attach_mcast(struct ib_qp *qp, union ib_gid *gid, u16 lid);
```

- `qp`: A handler of a UD QP to be attached to the multicast group.
- `gid`: The GID of the multicast group that the QP will be added to.
- `lid`: The LID of the multicast group that the QP will be added to.

The ib_detach_mcast() method

The `ib_detach_mcast()` method detaches a UD QP from a multicast group within an RDMA device. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_detach_mcast(struct ib_qp *qp, union ib_gid *gid, u16 lid);
```

- `qp`: A handler of a UD QP to be detached from the multicast group.
- `gid`: The GID of the multicast group that the QP will be removed from.
- `lid`: The LID of the multicast group that the QP will be removed from.

Completion Queue (CQ)

A Work Completion specifies that a corresponding Work Request was completed and provides some information. about it: its status, the used opcode, its size, and so on. A CQ is an object that consists of Work Completions.

The ib_create_cq() Method

The `ib_create_cq()` method creates a CQ. It will return a pointer to the newly created CQ on success or an ERR_PTR() which specifies the reason for the failure.

```
struct ib_cq *ib_create_cq(struct ib_device *device, ib_comp_handler comp_handler,
void (*event_handler)(struct ib_event *, void *), void *cq_context, int cqe, int comp_vector);
```

- `device`: The RDMA device that this CQ is being associated with.
- `comp_handler`: A pointer to a callback that will be called when a completion event occur to the CQ.
- `event_handler`: A pointer to a callback that will be called in case of an affiliated asynchronous event to the CQ.

- `cq_context`: A user-defined context that can be associated with the CQ.
- `cqe`: The requested number of Work Completions that this CQ can hold.
- `comp_vector`: The index of the RDMA device's completion vector to work on. If the IRQ affinity masks of these interrupts are spread across the cores, this value can be used to spread the completion workload over all of the cores.

The ib_resize_cq() Method

The `ib_resize_cq()` method changes the size of the CQ to hold at least the new size, either by increasing the CQ size or decreasing it. Even if the user asks to resize a CQ, its size may not be resized.

```
int ib_resize_cq(struct ib_cq *cq, int cqe);
```

- `cq`: The CQ to be resized. This value cannot be lower than the number of Work Completions that exists in the CQ.
- `cqe`: The requested number of Work Completions that this CQ can hold.

The ib_modify_cq() Method

The `ib_modify_cq()` method changes the moderation parameter for a CQ. A Completion event will be generated if at least a specific number of Work Completion will enter the CQ or a timeout will expire. Using it may help to reduce the number of interrupts that happen to the RDMA device. It will return 0 on success or the -errno value with the reason for the failure.

```
int ib_modify_cq(structib_cq *cq, u16 cq_count, u16 cq_period);
```

- `cq`: The CQ to be modified.
- `cq_count`: The number of Work Completions that will be added to the CQ, since the last Completion event, that will trigger a CQ event.
- `cq_period`: The number of microseconds that will pass, since the last Completion event, that will trigger a CQ event.

The ib_peek_cq() Method

The `ib_peek_cq()` method returns the number of available Work Completions in the CQ. If the number of Work Completions in the CQ is equal to or greater than `wc_cnt`, it will return `wc_cnt`. Otherwise it will return the actual number of the Work Completions in the CQ. If an error occurred, it will return the errno value with the reason for the failure.

```
int ib_peek_cq(structib_cq *cq, intwc_cnt);
```

- `cq`: The CQ to peek.
- `cq_count`: The number of Work Completions that will added to the CQ, since the last Completion event, that will trigger a CQ event.

The ib_req_notify_cq() Method

The `ib_req_notify_cq()` method requests that a Completion event notification be created. Its return value can be:

- 0: This means that the notification was requested successfully. If IB_CQ_REPORT_MISSED_EVENTS was used, then a return value of 0 means that there aren't any missed events.
- Positive value is returned only when IB_CQ_REPORT_MISSED_EVENTS is used and there are missed events. The user should call the `ib_poll_cq()` method in order to read the Work Completions that exist in the CQ.
- Negative value is returned when an error occurred. The –errno value is returned, specifying the reason for the failure.

```
static inline int ib_req_notify_cq(struct ib_cq *cq,
                      enum ib_cq_notify_flags flags);
```

- `cq`: The CQ that this Completion event will be generated for.
- `flags`: Information about the Work Completion that will cause the Completion event notification to be created. Can be one of:
  - IB_CQ_NEXT_COMP: The next Work Completion that will be added to the CQ, after calling this method, will trigger the CQ event.
  - IB_CQ_SOLICITED: The next Solicited Work Completion that will be added to the CQ, after calling this method, will trigger the CQ event.

Both of those values can be bitwise ORed with IB_CQ_REPORT_MISSED_EVENTS in order to request a hint about missed events (that is, when calling this method and there are already Work Completions in this CQ).

The ib_req_ncomp_notif() Method

The `ib_req_ncomp_notif()` method requests that a Completion event notification be created when the number of Work Completions in the CQ equals `wc_cnt`. It will return 0 on success, or the errno value with the reason for the failure.

```
static inline int ib_req_ncomp_notif(struct ib_cq *cq, int wc_cnt);
```

- `cq`: The CQ that this Completion event will be generated for.
- `wc_cnt`: The number of Work Completions that the CQ will hold before a Completion event notification is generated.

The ib_poll_cq() Method

The `ib_poll_cq()` method polls Work Completions from a CQ. It reads the Work Completion from the CQ and removes them. The Work Completions are read in the order they were added to the CQ. It will return 0 or a positive number to indicate the number of Work Completions that were read or the -errno value with the reason for the failure.

```
static inline int ib_poll_cq(struct ib_cq *cq, int num_entries,
               struct ib_wc *wc);
```

- `cq`: The CQ to be polled.

- `num_entries`: The maximum number of Work Completions to be polled.
- `wc`: An array that the number of polled Work Completions will be stored in.

The ib_wc Struct

Every Work Completion is represented by `struct ib_wc`:

```
struct ib_wc {
    u64            wr_id;
    enum ib_wc_status    status;
    enum ib_wc_opcode    opcode;
    u32            vendor_err;
    u32            byte_len;
    struct ib_qp          *qp;
    union {
        __be32        imm_data;
        u32        invalidate_rkey;
    } ex;
    u32            src_qp;
    int            wc_flags;
    u16            pkey_index;
    u16            slid;
    u8            sl;
    u8            dlid_path_bits;
    u8            port_num;
};
```

- `wr_id`: A 64-bit value that was associated with the corresponding Work Request.
- `status`: Status of the ended Work Request. Can be:
    - IB_WC_SUCCESS: Operation completed successfully.
    - IB_WC_LOC_LEN_ERR: Local length error. Either sent message is too big to be handled or incoming message is bigger than the available Receive Request.
    - IB_WC_LOC_QP_OP_ERR: Local QP operation error. An internal QP consistency error was detected while processing a Work Request.
    - IB_WC_LOC_EEC_OP_ERR: Local EE context operation error. Deprecated, since RD QPs aren’t supported.
    - IB_WC_LOC_PROT_ERR: Local protection error. The protection of the Work Request buffers is invalid to the requested operation.
    - IB_WC_WR_FLUSH_ERR: Work Request flushed error. The Work Request was completed when the QP was in the Error state.
    - IB_WC_MW_BIND_ERR: Memory Windows bind error. The operation of the Memory Windows binding failed.
    - IB_WC_BAD_RESP_ERR: Bad response error. Unexpected transport layer opcode returned by the responder.

- IB_WC_LOC_ACCESS_ERR: Local access error. A protection error occurred on local buffers during the processing of an RDMA Write With Immediate message.
- IB_WC_REM_INV_REQ_ERR: Remove invalid request error. The incoming message is invalid.
- IB_WC_REM_ACCESS_ERR: Remote access error. A protection error occurred to incoming RDMA operation.
- IB_WC_REM_OP_ERR: Remote operation error. The incoming operation couldn't be completed successfully.
- IB_WC_RETRY_EXC_ERR: Transport retry counter exceeded. The remote QP didn't send any Ack or Nack, and the timeout was expired after the message retransmission.
- IB_WC_RNR_RETRY_EXC_ERR: RNR retry exceeded. The RNR NACK return count was exceeded.
- IB_WC_LOC_RDD_VIOL_ERR: Local RDD violation error. Deprecated, since RD QPs aren't supported.
- IB_WC_REM_INV_RD_REQ_ERR: Remove invalid RD request. Deprecated, since RD QPs aren't supported.
- IB_WC_REM_ABORT_ERR: Remote aborted error. The responder aborted the operation.
- IB_WC_INV_EECN_ERR: Invalid EE Context number. Deprecated, since RD QPs aren't supported.
- IB_WC_INV_EEC_STATE_ERR: Invalid EE context state error. Deprecated, since RD QPs aren't supported.
- IB_WC_FATAL_ERR: Fatal error.
- IB_WC_RESP_TIMEOUT_ERR: Response timeout error.
- IB_WC_GENERAL_ERR: General error. Other error which isn't covered by one of the earlier errors.

- `opcode`: The operation of the corresponding Work Request that was ended with this Work Completion. Can be:
  - IB_WC_SEND: Send operation was completed in the sender side.
  - IB_WC_RDMA_WRITE: RDMA Write operation was completed in the sender side.
  - IB_WC_RDMA_READ: RDMA Read operation was completed in the sender side.
  - IB_WC_COMP_SWAP: Compare and Swap operation was completed in the sender side.
  - IB_WC_FETCH_ADD: Fetch and Add operation was completed in the sender side.
  - IB_WC_BIND_MW: Memory bind operation was completed in the sender side.
  - IB_WC_LSO: Send operation with Large Send Offload (LSO) was completed in the sender side.
  - IB_WC_LOCAL_INV: Local invalidate operation was completed in the sender side.
  - IB_WC_FAST_REG_MR: Fast registration operation was completed in the sender side.
  - IB_WC_MASKED_COMP_SWAP: Masked Compare and Swap operation was completed in the sender side.

    - IB_WC_MASKED_FETCH_ADD: Masked Fetch and Add operation was completed in the sender side.
    - IB_WC_RECV: Receive Request of an incoming send operation was completed in the receiver side.
    - IB_WC_RECV_RDMA_WITH_IMM: Receive Request of an incoming RDMA Write with immediate operation was completed in the receiver side.
- `vendor_err`: A vendor-specific value that provides extra information about the reason for the error.
- `byte_len`: If this is a Work Completion that was created from the end of a Receive Request, the `byte_len` value indicates the number of bytes that were received.
- `qp`: Handle of the QP that got the Work Completion. It is useful when QPs are associated with an SRQ—this way you can know the handle associated with the QP, that its incoming message consumed the Receive Request from the SRQ.
- `ex.imm_data`: Out Of Band data (32 bits), in network order, that was sent with the message. It is available if IB_WC_WITH_IMM is set in `wc_flags`.
- `ex.invalidate_rkey`: The `rkey` that was invalidated. It is available if IB_WC_WITH_INVALIDATE is set in `wc_flags`.
- `src_qp`: Source QP number. The QP number that sent this message. Only relevant for UD QPs.
- `wc_flags`: Flags that provide information about the Work Completion. It is a bitwise OR of the masks:
    - IB_WC_GRH: Indicator that the message was received has a GRH and the first 40 bytes of the Receive Request buffers contains it. Only relevant for UD QPs.
    - IB_WC_WITH_IMM: Indicator that the received message has immediate data.
    - IB_WC_WITH_INVALIDATE: Indicator that a Send with Invalidate message was received.
    - IB_WC_IP_CSUM_OK: Indicator that the received message passed the IP checksum test done by the RDMA device. This is available only if the RDMA device supports IP checksum offload. It is available if IB_DEVICE_UD_IP_CSUM is set in the device flags.
- `pkey_index`: The P_Key index, relevant only for GSI QPs.
- `slid`: The source LID of the message. Only relevant for UD QPs.
- `sl`: The Service Level of the message. Only relevant for UD QPs.
- `dlid_path_bits`: The destination LID path bits. Only relevant for UD QPs.
- `port_num`: The port number from which the message came in. Only relevant for Direct Route SMPs on switches.

The ib_destory_cq() Method

The `ib_destory_cq()` method destroys a CQ. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_destroy_cq(struct ib_cq *cq);
```

- `cq`: The CQ to be destroyed.

Queue Pair (QP)

QP is a resource that combines two Work Queues together: the Send Queue and the Receive Queue. Each queue acts as a FIFO. WRs that are being posted to each Work Queue will be processed by the order of their arrival. However, there isn't any guarantee about the order between the Queues. This resource is the resource that sends and receives packets.

The ib_qp_cap Struct

The QP's Work Queues sizes are represented by `struct ib_qp_cap`:

```
struct ib_qp_cap {
    u32    max_send_wr;
    u32    max_recv_wr;
    u32    max_send_sge;
    u32    max_recv_sge;
    u32    max_inline_data;
};
```

- `max_send_wr`: The maximum number of outstanding Work Requests that this QP can hold in the Send Queue.
- `max_recv_wr`: The maximum number of outstanding Work Requests that this QP can hold in the Receive Queue. This value is ignored if the QP is associated with an SRQ.
- `max_send_sge`: The maximum number of scatter/gather elements that each Work Request in the Send Queue will be able to hold.
- `max_recv_sge`: The maximum number of scatter/gather elements that each Work Request in the Receive Queue will be able to hold.
- `max_inline_data`: The maximum message size that can be sent inline.

The ib_create_qp() Method

The `ib_create_qp()` method creates a QP. It will return a pointer to the newly created QP on success or an ERR_PTR() which specifies the reason for the failure.

```
struct ib_qp *ib_create_qp(struct ib_pd *pd,
        struct ib_qp_init_attr *qp_init_attr);
```

- `pd`: The PD that this QP is being associated with.
- `qp_init_attr`: The attributes that this QP will be created with.

The ib_qp_init_attr Struct

The created QP attributes are represented by `struct ib_qp_init_attr`:

```
struct ib_qp_init_attr {
    void                    (*event_handler)(struct ib_event *, void *);
    void                *qp_context;
    struct ib_cq            *send_cq;
    struct ib_cq            *recv_cq;
    struct ib_srq            *srq;
    struct ib_xrcd           *xrcd;     /* XRC TGT QPs only */
    struct ib_qp_cap        cap;
    enum ib_sig_type        sq_sig_type;
    enum ib_qp_type        qp_type;
    enum ib_qp_create_flags    create_flags;
    u8                port_num; /* special QP types only */
};
```

- `event_handler`: A pointer to a callback that will be called in case of an affiliated asynchronous event to the QP.
- `qp_context`: User-defined context that can be associated with the QP.
- `send_cq`: A CQ that is being associated with the Send Queue of this QP.
- `recv_cq`: A CQ that is being associated with the Receive Queue of this QP.
- `srq`: A SRQ that is being associated with the Receive Queue of this QP or NULL if the QP isn't associated with an SRQ.
- `xrcd`: An XRC domain that this QP will be associated with. Relevant only if qp_type is IB_QPT_XRC_TGT.
- `cap`: A structure that describes the size of the Send and Receive Queues. This structure is described earlier.
- `sq_sig_type`: The signaling type of the Send Queue. It can be:
  - IB_SIGNAL_ALL_WR: Every posted Send Request to the Send Queue will end with a Work Completion.
  - IB_SIGNAL_REQ_WR: Only posted Send Requests to the Send Queue with an explicit request, i.e. set the IB_SEND_SIGNALED flag—will end with a Work Completion. This is called *selective signaling*.
- `qp_type`: The QP transport type. Can be:
  - IB_QPT_SMI: A Subnet Management Interface QP.
  - IB_QPT_GSI: A General Service Interface QP.
  - IB_QPT_RC: A Reliable Connected QP.
  - IB_QPT_UC: An Unreliable Connected QP.
  - IB_QPT_UD: An Unreliable Datagram QP.
  - IB_QPT_RAW_IPV6: An IPv6 raw datagram QP.

  - IB_QPT_RAW_ETHERTYPE: An EtherType raw datagram QP.
  - IB_QPT_RAW_PACKET: A raw packet QP.
  - IB_QPT_XRC_INI: An XRC-initiator QP.
  - IB_QPT_XRC_TGT: An XRC-target QP.
- `create_flags`: QP attributes flags. It is a bitwise OR of the masks:
  - IB_QP_CREATE_IPOIB_UD_LSO: The QP will be used to send IPoIB LSO messages.
  - IB_QP_CREATE_BLOCK_MULTICAST_LOOPBACK: Block loopback multicast packets.
- `port_num`: The RDMA device port number that this QP is associated with. Only relevant when `qp_type` is IB_QPT_SMI or IB_QPT_GS.

The ib_modify_qp() Method

The `ib_modify_qp()` method modifies the attributes of the QP. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_modify_qp(struct ib_qp *qp,
    struct ib_qp_attr *qp_attr,
    int qp_attr_mask);
```

- `qp`: The QP to be modified.
- `qp_attr`: The QP attributes, as described earlier.
- `qp_attr_mask`: The QP attributes to be changed. Each mask specifies the attributes that will be modified in this QP transition, such as specifying which attributes in `qp_attr` will be used. It is a bitwise OR of the masks:
  - IB_QP_STATE: Modifies the QP state, specified in the `qp_state` field.
  - IB_QP_CUR_STATE: Modifies the assumed current QP state, specified in the `cur_qp_state` field.
  - IB_QP_EN_SQD_ASYNC_NOTIFY: Modifies the status of the request for notification when the QP state is SQD.drained, specified in the `en_sqd_async_notify` field.
  - IB_QP_ACCESS_FLAGS: Modifies the allowed incoming Remote operations, specified in the `qp_access_flags` field.
  - IB_QP_PKEY_INDEX: Modifies the index in the P_Key table that this QP is associated with in the primary path, specified in the `pkey_index` field.
  - IB_QP_PORT: Modifies the RDMA device's port number that QP's primary path is associated with, specified in the `port_num` field.
  - IB_QP_QKEY: Modifies the Q-Key of the QP, specified in the `qkey` field.
  - IB_QP_AV: Modifies the Address Vector attributes of the QP, specified in the `ah_attr` field.
  - IB_QP_PATH_MTU: Modifies the MTU of the path, specified in the `path_mtu` field.
  - IB_QP_TIMEOUT: Modifies the timeout to wait before retransmission, specified in the field timeout.

- IB_QP_RETRY_CNT: Modifies the number of retries of the QP for lack of Ack/Nack, specified in the `retry_cnt` field.
- IB_QP_RNR_RETRY: Modifies the number of RNR retry of the QP, specified in the `rq_psn` field.
- IB_QP_RQ_PSN: Modifies the start PSN of the received packets, specified in the `rnr_retry` field.
- IB_QP_MAX_QP_RD_ATOMIC: Modifies the number of RDMA Read and Atomic operations that this QP can process in parallel as an initiator, specified in the `max_rd_atomic` field.
- IB_QP_ALT_PATH: Modifies the alternate path of the QP, specified in the `alt_ah_attr`, `alt_pkey_index`, `alt_port_num`, and `alt_timeout` fields.
- IB_QP_MIN_RNR_TIMER: Modifies the minimum RNR timer that the QP will report to the remote side in the RNR Nak, specified in the `min_rnr_timer` field.
- IB_QP_SQ_PSN: Modifies the start PSN of the sent packets, specified in the `sq_psn` field.
- IB_QP_MAX_DEST_RD_ATOMIC: Modifies the number of RDMA Read and Atomic operations that this QP can process in parallel as an initiator, specified in the `max_dest_rd_atomic` field.
- IB_QP_PATH_MIG_STATE: Modifies the state of the path migration state machine, specified in the `path_mig_state` field.
- IB_QP_CAP: Modifies the size of the Work Queues in the QP (both Send and Receive Queues), specified in the `cap` field.
- IB_QP_DEST_QPN: Modifies the destination QP number, specified in the `dest_qp_num` field.

The ib_qp_attr Struct

The QP attributes are represented by `struct ib_qp_attr`:

```
struct ib_qp_attr {
    enum ib_qp_state    qp_state;
    enum ib_qp_state    cur_qp_state;
    enum ib_mtu        path_mtu;
    enum ib_mig_state    path_mig_state;
    u32            qkey;
    u32            rq_psn;
    u32            sq_psn;
    u32            dest_qp_num;
    int            qp_access_flags;
    struct ib_qp_cap    cap;
    struct ib_ah_attr    ah_attr;
    struct ib_ah_attr    alt_ah_attr;
    u16            pkey_index;
    u16            alt_pkey_index;
    u8            en_sqd_async_notify;
    u8            sq_draining;
    u8            max_rd_atomic;
    u8              max_dest_rd_atomic;
    u8              min_rnr_timer;
    u8              port_num;
    u8              timeout;
    u8              retry_cnt;
    u8              rnr_retry;
    u8              alt_port_num;
    u8              alt_timeout;
};
```

- `qp_state`: The state to move the QP to. Can be:
    - IB_QPS_RESET: Reset state.
    - IB_QPS_INIT: Initialized state.
    - IB_QPS_RTR: Ready To Receive state.
    - IB_QPS_RTS: Ready To Send state.
    - IB_QPS_SQD: Send Queue Drained state.
    - IB_QPS_SQE: Send Queue Error state.
    - IB_QPS_ERR: Error state.
- `cur_qp_state`: The assumed current state of the QP. Can be like `qp_state`.
- `path_mtu`: The size of the MTU in the path. Can be:
    - IB_MTU_256: 256 bytes.
    - IB_MTU_512: 512 bytes.
    - IB_MTU_1024: 1,024 bytes.
    - IB_MTU_2048: 2,048 bytes.
    - IB_MTU_4096: 4,096 bytes.
- `path_mig_state`: The path migration state machine, used in APM (Automatic Path Migration). Can be:
    - IB_MIG_MIGRATED: Migrated. The state machine of path migration is Migrated (initial state of migration was done).
    - IB_MIG_REARM: Rearm. The state machine of path migration is Rearm (attempt to try to coordinate the remote RC QP to move both local and remote QPs to Armed state).
    - IB_MIG_ARMED: Armed. The state machine of path migration is Armed (both local and remote QPs are ready to perform a path migration).
- `qkey`: The Q_Key of the QP.
- `rq_psn`: The expected PSN of the first packet in the Receive Queue. The value is 24 bits.
- `sq_psn`: The used PSN of the first packet in the Send Queue. The value is 24 bits.
- `dest_qp_num`: The QP number in the remote (destination) side. The value is 24 bits.

- `qp_access_flags`: The allowed incoming RDMA and Atomic operations. It is a bitwise OR of the masks:
    - IB_ACCESS_REMOTE_WRITE: Incoming RDMA Write operations are allowed.
    - IB_ACCESS_REMOTE_READ: Incoming RDMA Read operations are allowed.
    - IB_ACCESS_REMOTE_ATOMIC: Incoming Atomic operations are allowed.
- `cap`: The QP size. The number of Work Requests in the Receive and Send Queues. This can be done only if the device supports QP resize—that is, the IB_DEVICE_RESIZE_MAX_WR is set in the device flags. This structure is described earlier.
- `ah_attr`: Address vector of the primary path of the QP. This structure is described earlier.
- `alt_ah_attr`: Address vector of the alternate path of the QP. This structure is described earlier.
- `pkey_index`: The P_Key index of the primary path that this QP is associated with.
- `alt_pkey_index`: The P_Key index of the alternate path that this QP is associated with.
- `en_sqd_async_notify`: If value isn't zero, request that the asynchronous event callback will be called when the QP will moved to SQE.drained state.
- `sq_draining`: Relevant only for `ib_query_qp()`. If value isn't zero, the QP is in state SQD. drainning (and not SQD.drained).
- `max_rd_atomic`: The number of RDMA Read and Atomic operations that this QP can process in parallel as an initiator.
- `max_dest_rd_atomic`: The number of RDMA Read and Atomic operations that this QP can process in parallel as a destination.
- `min_rnr_timer`: The timeout to wait before resend the message again if the remote side responds with an RNR Nack.
- `port_num`: The RDMA device's Port number that this QP is associated with in the Primary path.
- `timeout`: The timeout to wait before resending the message again if the remote side didn't respond with any Ack or Nack in the primary path. The `timeout` is a 5-bit value, 0 is infinite time, and any other value means that the timeout will be 4.096 * 2 ^ `timeout` usec.
- `retry_cnt`: The number of times to (re)send the message if the remote side didn't respond with any Ack or Nack.
- `rnr_retry`: The number of times to (re)send the message if the remote side answered with an RNR Nack. 3 bits value, 7 means infinite retry. The value can be:
    - IB_RNR_TIMER_655_36: Delay of 655.36 milliseconds.
    - IB_RNR_TIMER_000_01: Delay of 0.01 milliseconds.
    - IB_RNR_TIMER_000_02: Delay of 0.02 milliseconds.
    - IB_RNR_TIMER_000_03: Delay of 0.03 milliseconds.
    - IB_RNR_TIMER_000_04: Delay of 0.04 milliseconds.
    - IB_RNR_TIMER_000_06: Delay of 0.06 milliseconds.
    - IB_RNR_TIMER_000_08: Delay of 0.08 milliseconds.
    - IB_RNR_TIMER_000_12: Delay of 0.12 milliseconds.

    - IB_RNR_TIMER_000_16: Delay of 0.16 milliseconds.
    - IB_RNR_TIMER_000_24: Delay of 0.24 milliseconds.
    - IB_RNR_TIMER_000_32: Delay of 0.32 milliseconds.
    - IB_RNR_TIMER_000_48: Delay of 0.48 milliseconds.
    - IB_RNR_TIMER_000_64: Delay of 0.64 milliseconds.
    - IB_RNR_TIMER_000_96: Delay of 0.96 milliseconds.
    - IB_RNR_TIMER_001_28: Delay of 1.28 milliseconds.
    - IB_RNR_TIMER_001_92: Delay of 1.92 milliseconds.
    - IB_RNR_TIMER_002_56: Delay of 2.56 milliseconds.
    - IB_RNR_TIMER_003_84: Delay of 3.84 milliseconds.
    - IB_RNR_TIMER_005_12: Delay of 5.12 milliseconds.
    - IB_RNR_TIMER_007_68: Delay of 7.68 milliseconds.
    - IB_RNR_TIMER_010_24: Delay of 10.24 milliseconds.
    - IB_RNR_TIMER_015_36: Delay of 15.36 milliseconds.
    - IB_RNR_TIMER_020_48: Delay of 20.48 milliseconds.
    - IB_RNR_TIMER_030_72: Delay of 30.72 milliseconds.
    - IB_RNR_TIMER_040_96: Delay of 40.96 milliseconds.
    - IB_RNR_TIMER_061_44: Delay of 61.44 milliseconds.
    - IB_RNR_TIMER_081_92: Delay of 81.92 milliseconds.
    - IB_RNR_TIMER_122_88: Delay of 122.88 milliseconds.
    - IB_RNR_TIMER_163_84: Delay of 163.84 milliseconds.
    - IB_RNR_TIMER_245_76: Delay of 245.76 milliseconds.
    - IB_RNR_TIMER_327_68: Delay of 327.86 milliseconds.
    - IB_RNR_TIMER_491_52: Delay of 391.52 milliseconds.
- `alt_port_num`: The RDMA device's Port number that this QP is associated with in the alternate path.
- `alt_timeout`: The timeout to wait before resend the message again if the remote side didn't respond with any Ack or Nack in the alternate path. 5-bit value, 0 is infinite time, and any other value means that the timeout will be 4.096 * 2 ^ `timeout` usec.

The ib_query_qp() Method

The `ib_query_qp()` method queries for the current QP attributes. Some of the attributes in `qp_attr` may change in subsequent calls to `ib_query_qp()` the state fields. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_query_qp(struct ib_qp *qp, struct ib_qp_attr *qp_attr, int qp_attr_mask,
struct ib_qp_init_attr *qp_init_attr);
```

- qp: The QP to be queried.
- qp_attr: The QP attributes, as described earlier.
- qp_attr_mask: The mask of the mandatory attributes to query. Low-level drivers can use it as a hint for the fields to be queried, but they may also ignore it as well and fill the whole structure.
- qp_init_attr: The QP init attributes, as described earlier.

The ib_destory_qp() method destroys a QP. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_destroy_qp(struct ib_qp *qp);
```

- qp: The QP to be destroyed.

The ib_open_qp() Method

The ib_open_qp() method obtains a reference to an existing sharable QP among multiple processes. The process that created the QP may exit, allowing transfer of the ownership of the QP to another process. It will return a pointer to the sharable QP on success or an ERR_PTR() which specifies the reason for the failure.

```
struct ib_qp *ib_open_qp(struct ib_xrcd *xrcd, struct ib_qp_open_attr *qp_open_attr);
```

- xrcd: The XRC domain that the QP will be associated with.
- qp_open_attr: The attributes of the existing QP to be opened.

The ib_qp_open_attr Struct

The shared QP attributes are represented by struct ib_qp_open_attr:

```
struct ib_qp_open_attr {
    void              (*event_handler)(struct ib_event *, void *);
    void             *qp_context;
    u32           qp_num;
    enum ib_qp_type    qp_type;
};
```

- event_handler: A pointer to a callback that will be called in case of an affiliated asynchronous event to the QP.
- qp_context: User-defined context that can be associated with the QP.
- qp_num: The QP number that this QP will open.
- qp_type: QP transport type. Only IB_QPT_XRC_TGT is supported.

The ib_close_qp() Method

The ib_close_qp() method releases an external reference to a QP. The underlying shared QP won't be destroyed until all internal references that were acquired by the ib_open_qp() method are released. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_close_qp(struct ib_qp *qp);
```

- qp: The QP to be closed.

The ib_post_recv() Method

The ib_post_recv() method takes a linked list of Receive Requests and adds them to the Receive Queue for future processing. Every Receive Request is considered outstanding until a Work Completion is generated after its processing. It will return 0 on success or the errno value with the reason for the failure.

```
static inline int ib_post_recv(struct ib_qp *qp, struct ib_recv_wr *recv_wr, struct ib_recv_wr
**bad_recv_wr);
```

- qp: The QP that the Receive Requests will be posted to.
- recv_wr: A linked list of Receive Request to be posted.
- bad_recv_wr: If there was an error with the handling of the Receive Requests, this pointer will be filled with the address of the Receive Request that caused this error.

The ib_post_send() Method

The ib_post_send() method takes a linked list of Send Requests as an argument and adds them to the Send Queue for future processing. Every Send Request is considered outstanding until a Work Completion is generated after its processing. It will return 0 on success or the errno value with the reason for the failure.

```
static inline int ib_post_send(struct ib_qp *qp, struct ib_send_wr *send_wr, struct ib_send_wr
**bad_send_wr);
```

- qp: The QP that the Send Requests will be posted to.
- send_wr: A linked list of Send Requests to be posted.
- bad_send_wr: If there was an error with the handling of the Send Requests, this pointer will be filled with the address of the Send Request that caused this error.

The ib_send_wr Struct

The Send Request is represented by struct ib_send_wr:

```
struct ib_send_wr {
    struct ib_send_wr          *next;
    u64            wr_id;
    struct ib_sge        *sg_list;
    int            num_sge;
    enum ib_wr_opcode    opcode;
    int            send_flags;
    union {
        __be32        imm_data;
        u32        invalidate_rkey;
    } ex;
    union {
        struct {
            u64    remote_addr;
            u32    rkey;
        } rdma;
        struct {
            u64    remote_addr;
            u64    compare_add;
            u64    swap;
            u64    compare_add_mask;
            u64    swap_mask;
            u32    rkey;
        } atomic;
        struct {
            struct ib_ah     *ah;
            void           *header;
            int             hlen;
            int             mss;
            u32        remote_qpn;
            u32        remote_qkey;
            u16        pkey_index; /* valid for GSI only */
            u8        port_num;   /* valid for DR SMPs on switch only */
        } ud;
        struct {
            u64                iova_start;
            struct ib_fast_reg_page_list       *page_list;
            unsigned int           page_shift;
            unsigned int           page_list_len;
            u32                length;
            int                access_flags;
            u32                rkey;
        } fast_reg;
        struct {
            struct ib_mw                    *mw;
            /* The new rkey for the memory window. */
            u32                              rkey;
            struct ib_mw_bind_info           bind_info;
        } bind_mw;
    } wr;
    u32            xrc_remote_srq_num;    /* XRC TGT QPs only */
};
```

- next: A pointer to the next Send Request in the list or NULL, if this is the last Send Request.
- wr_id: 64-bit value that is associated with this Send Request and will be available in the corresponding Work Completion.
- sg_list: The array of the scatter/gather elements. As described earlier.
- num_sge: The number of entries in sg_list. The value zero means that the message size is zero bytes.

- opcode: The operation to perform. This affects the way that data is being transferred, the direction of it, and whether a Receive Request will be consumed in the remote side and which fields in the Send Request (`send_wr`) will be used. Can be:
    - IB_WR_RDMA_WRITE: RDMA Write operation.
    - IB_WR_RDMA_WRITE_WITH_IMM: RDMA Write with immediate operation.
    - IB_WR_SEND: Send operation.
    - IB_WR_SEND_WITH_IMM: Send with immediate operation.
    - IB_WR_RDMA_READ: RDMA Read operation.
    - IB_WR_ATOMIC_CMP_AND_SWP: Compare and Swap operation.
    - IB_WR_ATOMIC_FETCH_AND_ADD: Fetch and Add operation.
- IB_WR_LSO: Send an IPoIB message with LSO (let the RDMA device fragment the big SKBs to multiple MSS-sized packets).LSO is an optimization feature which allows to use large packets by reducing CPU overhead.
    - IB_WR_SEND_WITH_INV: Send with invalidate operation.
    - IB_WR_RDMA_READ_WITH_INV: RDMA Read with invalidate operation.
    - IB_WR_LOCAL_INV: Local invalidate operation.
    - IB_WR_FAST_REG_MR: Fast MR registration operation.
    - IB_WR_MASKED_ATOMIC_CMP_AND_SWP: Masked Compare and Swap operation.
    - IB_WR_MASKED_ATOMIC_FETCH_AND_ADD: Masked Fetch and Add operation.
    - IB_WR_BIND_MW: Memory bind operation.
- `send_flags`: Extra attributes for the Send Request. It is a bitwise OR of the masks:
    - IB_SEND_FENCE: Before performing this operation, wait until the processing of prior Send Requests has ended.
    - IB_SEND_SIGNALED: If the QP was created with selective signaling, when the processing of this Send Request is ended, a Work Completion will be generated.
    - IB_SEND_SOLICITED: Mark that a Solicited event will be created in the remote side.
    - IB_SEND_INLINE: Post this Send Request as inline—that is, let the low-level driver read the memory buffers in if `sg_list` instead of the RDMA device; this may increase the latency.
    - IB_SEND_IP_CSUM: Send an IPoIB message and calculate the IP checksum in HW (checksum offload).
- `ex.imm_data`: The immediate data to send. This value is relevant if `opcode` is IB_WR_SEND_WITH_IMM or IB_WR_RDMA_WRITE_WITH_IMM.
- `ex.invalidate_rkey`: The `rkey` to be invalidated. This value is relevant if opcode is IB_WR_SEND_WITH_INV.

The following union is relevant if opcode is IB_WR_RDMA_WRITE, IB_WR_RDMA_WRITE_WITH_IMM, or IB_WR_RDMA_READ:

- `wr.rdma.remote_addr`: The remote address that this Send Request is going to access.
- `wr.rdma.rkey`: The Remote Key (`rkey`) of the MR that this Send Request is going to access.

The following union is relevant if `opcode` is IB_WR_ATOMIC_CMP_AND_SWP, IB_WR_ATOMIC_FETCH_AND_ADD,IB_WR_MASKED_ATOMIC_CMP_AND_SWP, or IB_WR_MASKED_ATOMIC_FETCH_AND_ADD:

- `wr.atomic.remote_addr`: The remote address that this Send Request is going to access.
- `wr.atomic.compare_add`: If opcode is IB_WR_ATOMIC_FETCH_AND_ADD*, this is the value to add to the content of `remote_addr`. Otherwise, this is the value to compare the content of `remote_addr` with.
- `wr.atomic.swap`: The value to place in `remote_addr` if the value in it is equal to `compare_add`. This value is relevant if opcode is IB_WR_ATOMIC_CMP_AND_SWP or IB_WR_MASKED_ATOMIC_CMP_AND_SWP.
- `wr.atomic.compare_add_mask`: If opcode is IB_WR_MASKED_ATOMIC_FETCH_AND_ADD, this is the mask of the values to change when adding the value of `compare_add` to the content of `remote_addr`. Otherwise, this is the mask to use on the content of `remote_addr` when comparing it with swap.
- `wr.atomic.swap_mask`: This is the mask of the value in the content of `remote_addr` to change. Relevant only if opcode is IB_WR_MASKED_ATOMIC_CMP_AND_SWP.
- `wr.atomic.rkey`: The `rkey` of the MR that this Send Request is going to access.

The following union is relevant if the QP type that this Send Request is being posted to is UD:

- `wr.ud.ah`: The address handle that describes the path to the target node(s).
- `wr.ud.header`: A pointer that contains the header. Relevant if opcode is IB_WR_LSO.
- `wr.ud.hlen`: The length of `wr.ud.header`. Relevant if opcode is IB_WR_LSO.
- `wr.ud.mss`: The Maximum Segment Size that the message will be fragmented to. Relevant if opcode is IB_WR_LSO.
- `wr.ud.remote_qpn`: The remote QP number to send the message to. The enumeration IB_MULTICAST_QPN should be used if sending this message to a multicast group.
- `wr.ud.remote_qkey`: The remote Q_Key value to use. If the MSB of this value is set, then the value of the Q_Key will be taken from the QP attributes.
- `wr.ud.pkey_index`: The P_Key index that the message will be sent with. Relevant if QP type is IB_QPT_GSI.
- `wr.ud.port_num`: The port number that the message will be sent from. Relevant for Direct Route SMP on a switch.

The following union is relevant if opcode is IB_WR_FAST_REG_MR:

- `wr.fast_reg.iova_start`: I/O Virtual Address of the newly created FMR.
- `wr.fast_reg.page_list`: List of pages to allocate to map in the FMR.
- `wr.fast_reg.page_shift`: Log 2 of size of "pages" to be mapped.
- `wr.fast_reg.page_list_len`: The number of pages in `page_list`.

- wr.fast_reg.length: The size, in bytes, of the FMR.
- wr.fast_reg.access_flags: The allowed operations on this FMR.
- wr.fast_reg.rkey: The value of the remote key to be assigned to the FMR.

The following union is relevant if opcode is IB_WR_BIND_MW:

- wr.bind_mw.mw: The MW to be bounded.
- wr.bind_mw.rkey: The value of the remote key to be assigned to the MW.
- wr.bind_mw.bind_info: The bind attributes, as explained in the next section.

The following member is relevant if the QP type that this Send Request is being posted to is XRCTGT:

- xrc_remote_srq_num: The remote SRQ that will receive the messages.

The ib_mw_bind_info Struct

The MW binding attributes for both MW type 1 and type 2 are represented by struct ib_mw_bind_info.

```
struct ib_mw_bind_info {
    struct ib_mr       *mr;
    u64        addr;
    u64        length;
    int        mw_access_flags;
};
```

- mr: A Memory Region that this Memory Window will be bounded to.
- addr: The address where the Memory Window will start from.
- length: The length, in bytes, of the Memory Window.
- mw_access_flags: The allowed incoming RDMA and Atomic operations. It is a bitwise OR of the masks:
    - IB_ACCESS_REMOTE_WRITE: Incoming RDMA Write operations are allowed.
    - IB_ACCESS_REMOTE_READ: Incoming RDMA Read operations are allowed.
    - IB_ACCESS_REMOTE_ATOMIC: Incoming Atomic operations are allowed.

Memory Windows (MW)

Memory Windows are used as a lightweight operation to change the allowed permission of incoming remote operations and invalidate them.

The ib_alloc_mw() Method

The ib_alloc_mw() method allocates a Memory Window. It will return a pointer to the newly allocated MW on success or an ERR_PTR() which specifies the reason for the failure.

```
struct ib_mw *ib_alloc_mw(struct ib_pd *pd, enum ib_mw_type type);
```

- `pd`: The PD that this MW is being associated with.
- `type`: The type of the Memory Window. Can be:
    - IB_MW_TYPE_1: MW that can be bounded using a verb and supports only association of a PD.
    - IB_MW_TYPE_2: MW that can be bounded using Work Request and supports association of a QP number only or a QP number and a PD.

The ib_bind_mw() Method

The `ib_bind_mw()` method binds a Memory Window to a specified Memory Region with a specific address, size, and remote permissions. If there isn't any immediate error, the `rkey` of the MW will be updated to the new value, but the bind operation may still fail asynchronously (and end with completion with error). It will return 0 on success or the errno value with the reason for the failure.

```
static inline int ib_bind_mw(struct ib_qp *qp, struct ib_mw *mw, struct ib_mw_bind *mw_bind);
```

- `qp`: The QP that the bind WR will be posted to.
- `mw`: The MW to bind.
- `mw_bind`: The bind attributes, as explained next.

The ib_mw_bind Struct

The MW binding attributes for type 1 MW are represented by `struct ib_mw_bind`.

```
struct ib_mw_bind {
    u64                    wr_id;
    int                    send_flags;
    struct ib_mw_bind_info bind_info;
};
```

- `wr_id`: A 64-bit value that is associated with this bind Send Request The value of Work Request id (`wr_id`) will be available in the corresponding Work Completion.
- `send_flags`: Extra attribute for the bind Send Request, as explained earlier. Only IB_SEND_FENCE and IB_SEND_SIGNALED are supported here.
- `bind_info`: More attributes for the bind operation. As explained earlier.

The ib_dealloc_mw() Method

The `ib_dealloc_mw()` method deallocates an MW. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_dealloc_mw(struct ib_mw *mw);
```

- `mw`: The MW to be deallocated.

Memory Region (MR)

Every memory buffer that is being accessed by the RDMA device needs to be registered. During the registration process, the memory will be pinned (prevented from being swapped out), and the memory translation information (from virtual addresses ➤ physical addresses) will be saved in the RDMA device. After the registration, every Memory Region has two keys: one for local access and one for remote access. Those keys will be used when specifying those memory buffers in Work Requests.

The ib_get_dma_mr() Method

The `ib_get_dma_mr()` method returns a Memory Region for system memory that is usable for DMA. Creating this MR isn't enough, and the `ib_dma_*()` methods below are needed in order to create or destroy addresses that the `lkey` and `rkey` of this MR will be used with. It will return a pointer to the newly allocated MR on success or an ERR_PTR() which specifies the reason for the failure.

```
struct ib_mr *ib_get_dma_mr(struct ib_pd *pd, int mr_access_flags);
```

- `pd`: The PD that this MR is being associated with.
- `mr_access_flags`: The allowed operations on this MR. Local Write is always supported in this MR. It is a bitwise OR of the masks:
  - IB_ACCESS_LOCAL_WRITE: Local write to this Memory Region is allowed.
  - IB_ACCESS_REMOTE_WRITE: Incoming RDMA Write operations to this Memory Region are allowed.
  - IB_ACCESS_REMOTE_READ: Incoming RDMA Read operations to this Memory Region are allowed.
  - IB_ACCESS_REMOTE_ATOMIC: Incoming Atomic operations to this Memory Region are allowed.
  - IB_ACCESS_MW_BIND: MW bind to this Memory Region is allowed.
  - IB_ZERO_BASED: Indication that the Virtual address is zero based.

The ib_dma_mapping_error() Method

The `ib_dma_mapping_error()` method checks if the DMA address that was returned from `ib_dma_*()` failed. It will return a non-zero value if there was any failure and zero if the operation finished successfully.

```
static inline int ib_dma_mapping_error(struct ib_device *dev, u64 dma_addr);
```

- dev: The RDMA device for which the DMA address was created by using an `ib_dma_*()` method.
- `dma_addr`: The DMA address to verify.

The ib_dma_map_single() Method

The `ib_dma_map_single()` method maps a kernel virtual address to a DMA address. It will return a DMA address that needed to be checked with the `ib_dma_mapping_error()` method for errors:

```
static inline u64 ib_dma_map_single(struct ib_device *dev, void *cpu_addr, size_t size, enum dma_
data_direction direction);
```

- dev: The RDMA device on which the DMA address will be created.
- cpu_addr: The kernel virtual address to map for DMA.
- size: The size, in bytes, of the region to map.
- direction: The direction of the DMA. Can be:
    - DMA_TO_DEVICE: DMA from the main memory to the device.
    - DMA_FROM_DEVICE: DMA from the device to main memory.
    - DMA_BIDIRECTIONAL: DMA from the main memory to the device or from the device to main memory.

The ib_dma_unmap_single() Method

The ib_dma_unmap_single() method unmaps a DMA mapping that was assigned using ib_dma_map_single():

```
static inline void ib_dma_unmap_single(struct ib_device *dev, u64 addr, size_t size, enum dma_data_
direction direction);
```

- dev: The RDMA device on which the DMA address was created.
- addr: The DMA address to unmap.
- size: The size, in bytes, of the region to unmap. This value must be the same value that was used in the ib_dma_map_single() method.
- direction: The direction of the DMA. This value must be the same value that was used in the ib_dma_map_single() method.

The ib_dma_map_single_attrs() Method

The ib_dma_map_single_attrs() method maps a kernel virtual address to a DMA address according to a DMA attributes. It will return a DMA address that is needed to be checked with the ib_dma_mapping_error() method for errors.

```
static inline u64 ib_dma_map_single_attrs(struct ib_device *dev, void *cpu_addr, size_t size, enum
dma_data_direction direction, struct dma_attrs *attrs);
```

- dev: The RDMA device on which the DMA address will be created.
- cpu_addr: The kernel virtual address to map for DMA.
- size: The size, in bytes, of the region to map.
- direction: The direction of the DMA. As described earlier.
- attrs: The DMA attributes for the mapping. If this value is NULL, this method behaves like the ib_dma_map_single() method.

The ib_dma_unmap_single_attrs() Method

The ib_dma_unmap_single_attrs() method unmaps a DMA mapping that was assigned using the ib_dma_map_
single_attrs() method:

```
static inline void ib_dma_unmap_single_attrs(struct ib_device *dev, u64 addr, size_t size,
enum dma_data_direction direction, struct dma_attrs *attrs);
```

- dev: The RDMA device on which the DMA address was created.
- addr: The DMA address to unmap.
- size: The size, in bytes, of the region to unmap. This value must be the same value that was used in the ib_dma_map_single_attrs() method.
- direction: The direction of the DMA. This value must be the same value that was used in the ib_dma_map_single_attrs() method.
- attrs: The DMA attributes of the mapping. This value must be the same value that was used in the ib_dma_map_single_attrs() method. If this value is NULL, this method behaves like the ib_dma_unmap_single() method.

The ib_dma_map_page() Method

The ib_dma_map_page() method maps a physical page to a DMA address. It will return a DMA address that needs to be checked with the ib_dma_mapping_error() method for errors:

```
static inline u64 ib_dma_map_page(struct ib_device *dev, struct page *page, unsigned long offset,
size_t size, enum dma_data_direction direction);
```

- dev: The RDMA device on which the DMA address will be created.
- page: The physical page address to map for DMA.
- offset: The offset within the page that the registration will start from.
- size: The size, in bytes, of the region.
- direction: The direction of the DMA. As described earlier.

The ib_dma_unmap_page() Method

The ib_dma_unmap_page() method unmaps a DMA mapping that was assigned using the ib_dma_map_page() method:

```
static inline void ib_dma_unmap_page(struct ib_device *dev, u64 addr, size_t size, enum dma_data_
direction direction);
```

- dev: The RDMA device on which the DMA address was created.
- addr: The DMA address to unmap.
- size: The size, in bytes, of the region to unmap. This value must be the same value that was used in the ib_dma_map_page() method.
- direction: The direction of the DMA. This value must be the same value that was used in the ib_dma_map_page() method.

The ib_dma_map_sg() Method

The `ib_dma_map_sg()` method maps a scatter/gather list to a DMA address. It will return a non-zero value on success and 0 on a failure.

```
static inline int ib_dma_map_sg(struct ib_device *dev, struct scatterlist *sg, int nents, enum dma_
data_direction direction);
```

- `dev`: The RDMA device on which the DMA address will be created.
- `sg`: An array of the scatter/gather entries to map.
- `nents`: The number of scatter/gather entries in `sg`.
- `direction`: The direction of the DMA. As described earlier.

The ib_dma_unmap_sg() Method

The `ib_dma_unmap_sg()` method unmaps a DMA mapping that was assigned using the `ib_dma_map_sg()` method:

```
static inline void ib_dma_unmap_sg(struct ib_device *dev, struct scatterlist *sg, int nents, enum
dma_data_direction direction);
```

- `dev`: The RDMA device on which the DMA address was created.
- `sg`: An array of the scatter/gather entries to unmap. This value must be the same value that was used in the `ib_dma_map_sg()` method.
- `nents`: The number of scatter/gather entries in `sg`. This value must be the same value that was used in the `ib_dma_map_sg()` method.
- `direction`: The direction of the DMA. This value must be the same value that was used in the `ib_dma_map_sg()` method.

The ib_dma_map_sg_attr() Method

The `ib_dma_map_sg_attr()` method maps a scatter/gather list to a DMA address according to a DMA attributes. It will return a non-zero value on success and 0 on a failure.

```
static inline int ib_dma_map_sg_attrs(struct ib_device *dev, struct scatterlist *sg, int nents, enum
dma_data_direction direction, struct dma_attrs *attrs);
```

- `dev`: The RDMA device on which the DMA address will be created.
- `sg`: An array of the scatter/gather entries to map.
- `nents`: The number of scatter/gather entries in `sg`.
- `direction`: The direction of the DMA. As described earlier.
- `attrs`: The DMA attributes for the mapping. If this value is NULL, this method behaves like the `ib_dma_map_sg()` method.

The ib_dma_unmap_sg() Method

The `ib_dma_unmap_sg()` method unmaps a DMA mapping that was done using the `ib_dma_map_sg()` method:

```
static inline void ib_dma_unmap_sg_attrs(struct ib_device *dev, struct scatterlist *sg, int nents,
enum dma_data_direction direction, struct dma_attrs *attrs);
```

- `dev`: The RDMA device on which the DMA address was created.
- `sg`: An array of the scatter/gather entries to unmap. This value must be the same value that was used in the `ib_dma_map_sg_attrs()` method.
- `nents`: The number of scatter/gather entries in `sg`. This value must be the same value that was used in the `ib_dma_map_sg_attrs()` method.
- `direction`: The direction of the DMA. This value must be the same value that was used in the `ib_dma_map_sg_attrs()` method.
- `attrs`: The DMA attributes of the mapping. This value must be the same value that was used in the `ib_dma_map_sg_attrs()` method. If this value is NULL, this method behaves like the `ib_dma_unmap_sg()` method.

The ib_sg_dma_address() Method

The `ib_sg_dma_address()` method returns the DMA address from a scatter/gather entry.

```
static inline u64 ib_sg_dma_address(struct ib_device *dev, struct scatterlist *sg);
```

- `dev`: The RDMA device on which the DMA address was created.
- `sg`: A scatter/gather entry.

The ib_sg_dma_len() Method

The `ib_sg_dma_len()` method returns the DMA length from a scatter/gather entry.

```
static inline unsigned int ib_sg_dma_len(struct ib_device *dev, struct scatterlist *sg);
```

- `dev`: The RDMA device on which the DMA address was created.
- `sg`: A scatter/gather entry.

The ib_dma_sync_single_for_cpu() Method

The `ib_dma_sync_single_for_cpu()` method transfers a DMA region ownership to the CPU. This method must be called before the CPU accesses a DMA-mapped buffer in order to read or modify its content, and prevents the device from accessing it:

```
static inline void ib_dma_sync_single_for_cpu(struct ib_device *dev, u64 addr, size_t size,
enum dma_data_direction dir);
```

- `dev`: The RDMA device on which the DMA address was created.
- `addr`: The DMA address to sync.

- size: The size, in bytes, of the region.
- direction: The direction of the DMA. As described earlier.

The ib_dma_sync_single_for_device() Method

The ib_dma_sync_single_for_device() method transfers a DMA region ownership to the device. This method must be called before the device can access a DMA-mapped buffer again after the ib_dma_sync_single_for_cpu() method was called.

```
static inline void ib_dma_sync_single_for_device(struct ib_device *dev, u64 addr, size_t size, enum
dma_data_direction dir);
```

- dev: The RDMA device on which the DMA address was created.
- addr: The DMA address to sync.
- size: The size, in bytes, of the region.
- direction: The direction of the DMA. As described earlier.

The ib_dma_alloc_coherent() Method

The ib_dma_alloc_coherent() method allocates a memory block that can be accessible by the CPU and maps it for DMA. It will return the virtual address that the CPU can access on success or NULL in case of a failure:

```
static inline void *ib_dma_alloc_coherent(struct ib_device *dev, size_t size, u64 *dma_handle, gfp_t flag);
```

- dev: The RDMA device on which the DMA address will be created.
- size: The size, in bytes, of the memory to allocate and map.
- direction: The direction of the DMA. As described earlier.
- dma_handle: A pointer that will be filled with the DMA address of the region, if the allocation succeeds.
- flag: Memory allocation flags. Can be:
  - GFP_KERNEL: To allow blocking (not in interrupt, not holding SMP locks).
  - GFP_ATOMIC: Prevent blocking.

The ib_dma_free_coherent() method

The ib_dma_free_coherent() method frees a memory block that was allocated using the ib_dma_alloc_coherent() method:

```
static inline void ib_dma_free_coherent(struct ib_device *dev, size_t size, void *cpu_addr,
u64 dma_handle);
```

- dev: The RDMA device on which the DMA address was created.
- size: The size, in bytes, of the memory region. This value must be the same value that was used in the ib_dma_alloc_coherent() method.

- cpu_addr: The CPU memory address to free. This value must be the value that was returned by the ib_dma_alloc_coherent() method.
- dma_handle: The DMA address to free. This value must be the value that was returned by the ib_dma_alloc_coherent() method.

The ib_reg_phys_mr() Method

The ib_reg_phys_mr() method takes a set of physical pages, register them and prepare a virtual address that can be accessed by an RDMA device. It will return a pointer to the newly allocated MR on success or an ERR_PTR(), which specifies the reason for the failure.

```
struct ib_mr *ib_reg_phys_mr(struct ib_pd *pd, struct ib_phys_buf *phys_buf_array, int num_phys_buf,
int mr_access_flags, u64 *iova_start);
```

- pd: The PD that this MR is being associated with.
- phys_buf_array: An array of physical buffers to use in the Memory Region.
- num_phys_buf: The number of physical buffers in phys_buf_array.
- mr_access_flags: The allowed operations on this MR. As specified earlier.
- iova_start: A pointer to the requested I/O Virtual Address to be associated with the Region, which is allowed to begin anywhere within the first physical buffer. The RDMA device will set this value with the actual I/O virtual address of the Region. This value may be different from the requested one.

The ib_phys_buf Struct

The physical buffer is represented by struct ib_phys_buf.

```
struct ib_phys_buf {
    u64      addr;
    u64      size;
};
```

- addr: The physical address of the buffer.
- size: The size of the buffer.

The ib_rereg_phys_mr() Method

The ib_rereg_phys_mr() method modifies the attributes of an existing Memory Region. This method can be thought of as a call to the ib_dereg_mr() method, which was followed by a call to the ib_reg_phys_mr() method. Where possible, resources are reused instead of being deallocated and reallocated. It will return 0 on success or the errno value with the reason for the failure:

```
int ib_rereg_phys_mr(struct ib_mr *mr, int mr_rereg_mask, struct ib_pd *pd, struct ib_phys_buf
*phys_buf_array, int num_phys_buf, int mr_access_flags, u64 *iova_start);
```

- mr: The Memory Region to be reregistered.
- mr_rereg_mask: The Memory Region attributes to be changed. It is a bitwise OR of the masks:
    - IB_MR_REREG_TRANS: Modify the memory pages of this Memory Region.
    - IB_MR_REREG_PD: Modify the PD of this Memory Region.
    - IB_MR_REREG_ACCESS: Modify the allowed operations of this Memory Region.
- pd: The new Protection Domain that this Memory Region will be associated with.
- phys_buf_array: The new physical pages to be used.
- num_phys_buf: The number of physical pages to be used.
- mr_access_flags: The new allowed operations of this Memory Region.
- iova_start: The new I/O Virtual Address of this Memory Region.

The ib_query_mr() Method

The ib_query_mr() method retrieves the attributes of a specific MR. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_query_mr(struct ib_mr *mr, struct ib_mr_attr *mr_attr);
```

- mr: The MR to be queried.
- mr_attr: The MR attributes as describe in the next section.

The MR attributes are represented by struct ib_mr_attr.

The ib_mr_attr Struct

```
struct ib_mr_attr {
    struct ib_pd    *pd;
    u64         device_virt_addr;
    u64         size;
    int         mr_access_flags;
    u32         lkey;
    u32         rkey;
};
```

- pd: The PD that the MR is associated with.
- device_virt_addr: The address of the virtual block that this MR covers.
- size: The size, in bytes, of the Memory Region.
- mr_access_flags: The access permissions of this Memory Region.
- lkey: The local key of this Memory Region.
- rkey: The remote key of this Memory Region.

The ib_dereg_mr() Method

The `ib_dereg_mr()` method deregisters an MR. This method may fail if a Memory Window is bounded to it. It will return 0 on success or the errno value with the reason for the failure:

```
int ib_dereg_mr(struct ib_mr *mr);
```

- `mr`: The MR to be deregistered.