Linux Kernel Training. Lecture 23

Network device drivers

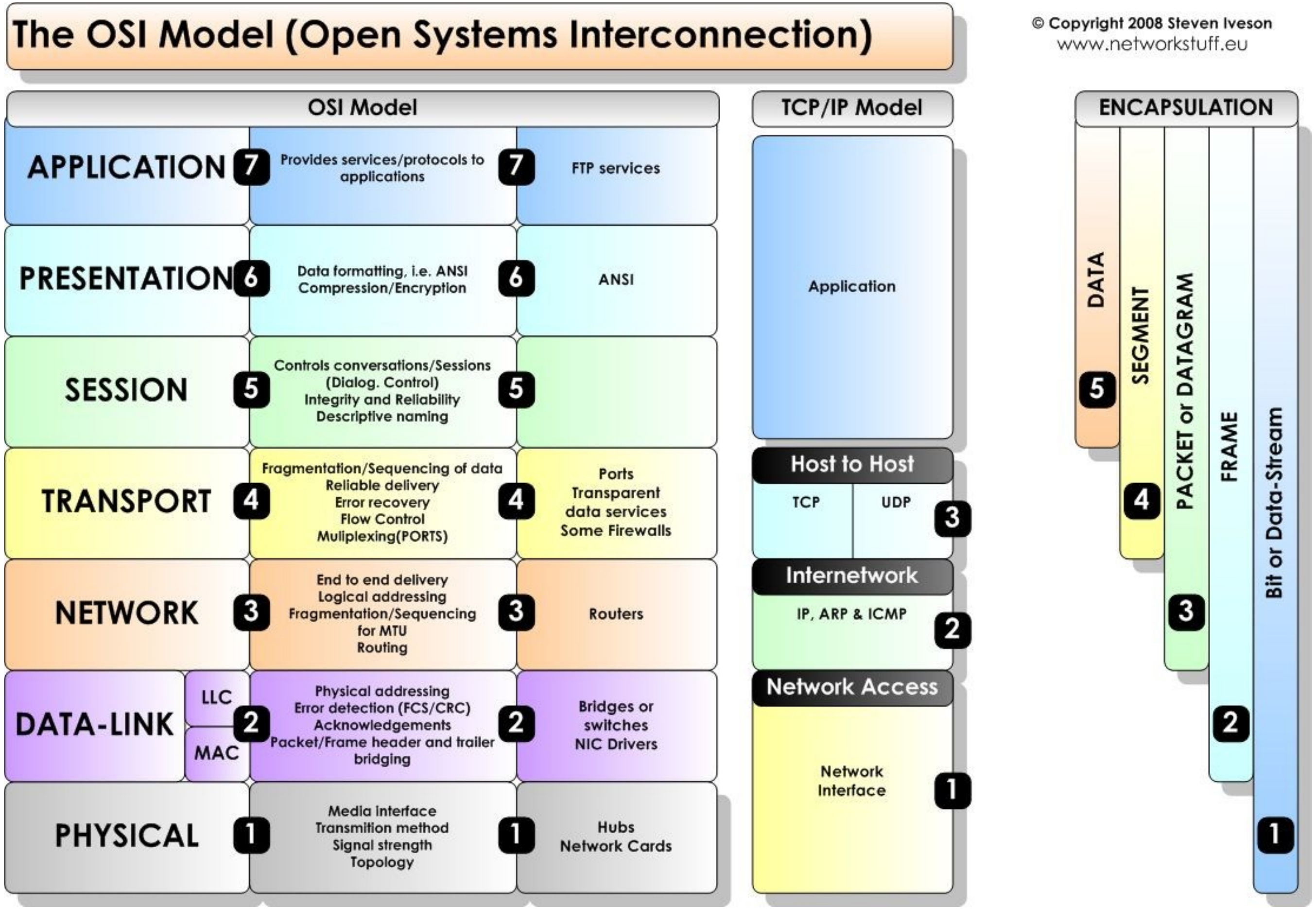
Oleksandr Redchuk
<oleksandr.redchuk@gmail.com>

June 15, 2020. GlobalLogic

Agenda

- OSI
- Hardware overview
- Kernel socket API
- Socket buffer
- Net device
- OAPI vs NAPI
- Code example (eth_dummy)

OSI model


The OSI Model (Open Systems Interconnection)
© Copyright 2008 Steven Iveson
www.networkstuff.eu
OSI Model
APPLICATION 7
Provides services/protocols to applications
7 FTP services
PRESENTATION 6
Data formatting, i.e. ANSI Compression/Encryption
6 ANSI
SESSION 5
Controls conversations/Sessions (Dialog. Control) Integrity and Reliability Descriptive naming
5
TRANSPORT 4
Fragmentation/Sequencing of data Reliable delivery Error recovery Flow Control Muliplexing(PORTS)
4 Ports Transparent data services Some Firewalls
NETWORK 3
End to end delivery Logical addressing Fragmentation/Sequencing for MTU Routing
3 Routers
DATA-LINK
LLC
MAC
2
Physical addressing Error detection (FCS/CRC) Acknowledgements Packet/Frame header and trailer bridging
2 Bridges or switches NIC Drivers
PHYSICAL 1
Media interface Transmition method Signal strength Topology
1 Hubs Network Cards
TCP/IP Model
Application
Host to Host
TCP
UDP
3
Internetwork
IP, ARP & ICMP
2
Network Access
Network Interface
1
ENCAPSULATION
DATA 5
SEGMENT 4
PACKET or DATAGRAM 3
FRAME 2
Bit or Data-Stream 1

OSI model

HTTP, FTP, …

MIME, …

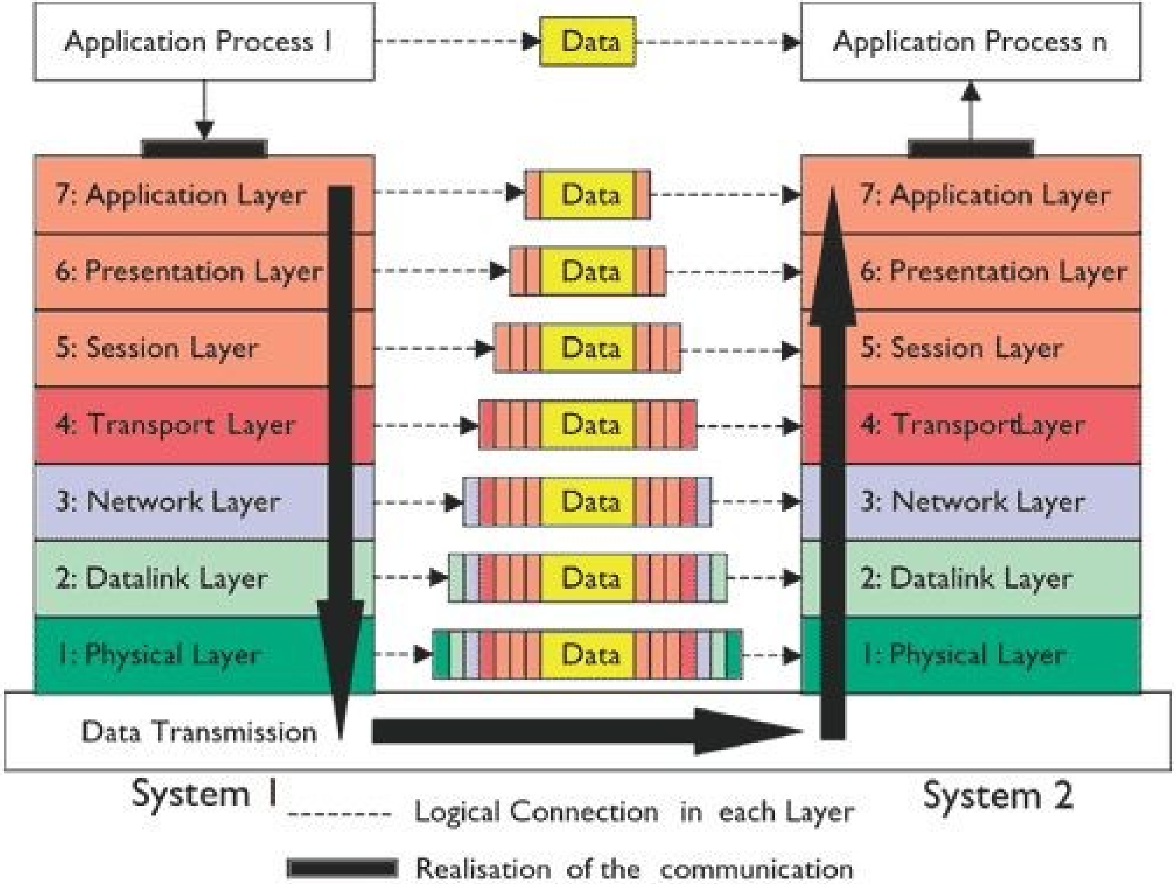
SOCKS, …

TCP, UDP, …

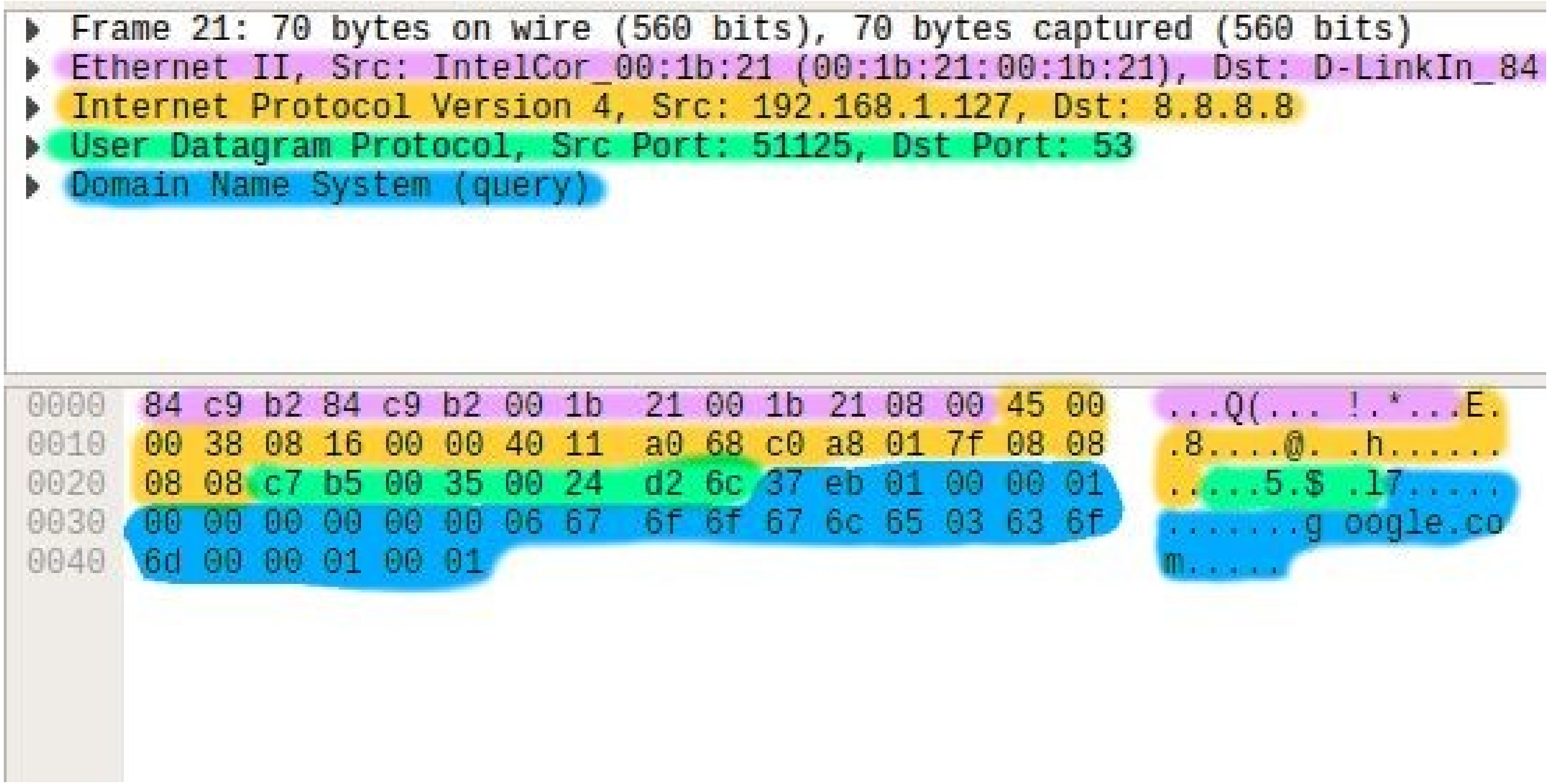
IP (v4, v6), …

IEEE802.3, …

IEEE802.3, …



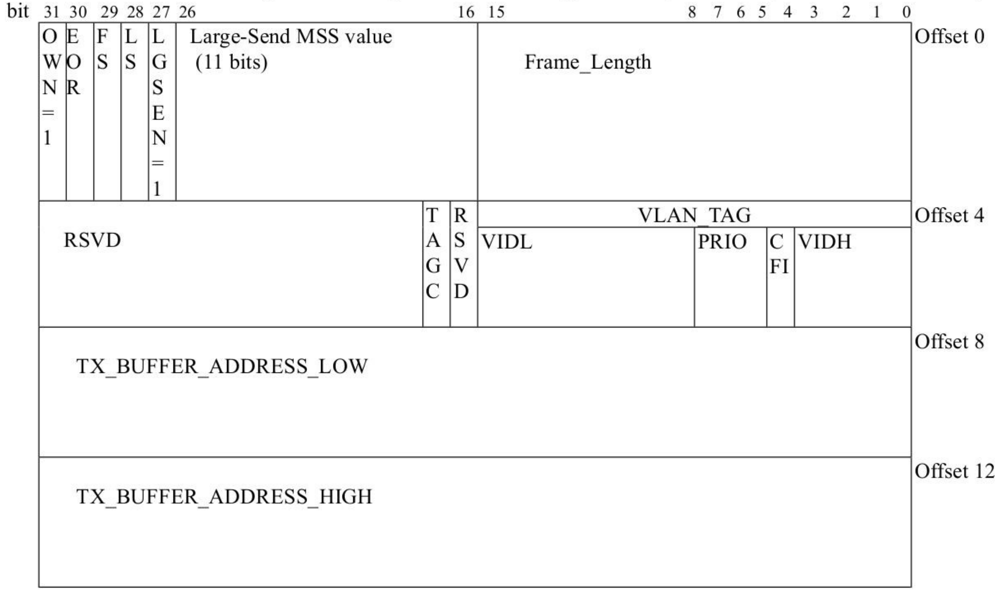
Network protocols



NIC hardware

- PHY and MAC
- General registers, MMIO
- Interrupts, DMA
- RX, TX descriptors
- Offloading:
  - Checksumming
  - Filtering
  - Classification
  - IRQ / DMA coalescing
  - Scatter-gather / Zero-copy
  - Timestamping (PTP, IEEE1588)
- See also: ethtool -h

Large-Send Task Offload Tx Descriptor Format (before transmitting, OWN=1, LGSEN=1, Tx command mode 0)



Offset#	Bit#	Symbol	Description
0	31	OWN	**Ownership:** This bit, when set, indicates that the descriptor is owned by THE NIC, and the data relative to this descriptor is ready to be transmitted. When cleared, it indicates that the descriptor is owned by host system. The NIC clears this bit when the relative buffer data is transmitted. In this case, OWN=1.
0	30	EOR	**End of Descriptor Ring:** This bit, when set, indicates that this is the

Kernel sockets

```
/* linux/net.h */
struct socket {
     socket_state       state;   /* socket state (SS_CONNECTED, etc) */
     short              type;    /* socket type (SOCK_STREAM, etc) */
     unsigned long      flags;   /* socket flags (SOCK_NOSPACE, etc) */
     struct socket_wq   *wq;     /* wait queue for several uses */
     struct file        *file;   /* File back pointer for gc */
     struct sock        *sk;     /* internal networking protocol agnostic representation */
     const struct proto_ops   *ops; /* protocol specific socket operations */
};

int sock_create_kern(struct net *net, int family, int type, int proto, struct socket **res);
int sock_create(int family, int type, int proto, struct socket **res);

struct net_proto_family {
     int       family;
     int       (*create)(struct net *net, struct socket *sock, int protocol, int kern);
     struct module  *owner;
};
```

Kernel sockets

```
/* linux/net.h */

int kernel_sendmsg(struct socket *sock, struct msghdr *msg,
               struct kvec *vec, size_t num, size_t len);
int kernel_recvmsg(struct socket *sock, struct msghdr *msg,
               struct kvec *vec, size_t num, size_t len, int flags);
int kernel_bind(struct socket *sock, struct sockaddr *addr, int addrlen);
int kernel_listen(struct socket *sock, int backlog);
int kernel_accept(struct socket *sock, struct socket **newsock, int flags);
int kernel_connect(struct socket *sock, struct sockaddr *addr, int addrlen, int flags);
int kernel_getsockname(struct socket *sock, struct sockaddr *addr, int *addrlen);
int kernel_getpeername(struct socket *sock, struct sockaddr *addr, int *addrlen);
int kernel_getsockopt(struct socket *sock, int level, int optname, char *optval, int *optlen);
int kernel_setsockopt(struct socket *sock, int level, int optname, char *optval, int optlen);
int kernel_sendpage(struct socket *sock, struct page *page, int offset,
               size_t size, int flags);
int kernel_sock_ioctl(struct socket *sock, int cmd, unsigned long arg);
int kernel_sock_shutdown(struct socket *sock, enum sock_shutdown_cmd how);
```

Kernel sockets

```
/* net/sock.h */

struct sock {
     ...
     socket_lock_t       sk_lock;
     ...
     struct {
          atomic_t  rmem_alloc;
          int       len;
          struct sk_buff *head;
          struct sk_buff *tail;
     } sk_backlog;
     ...
     struct sk_buff_head sk_receive_queue;
     struct sk_buff_head sk_write_queue;
     ...
};
```

Socket buffer

```
/* linux/skbuff.h */

struct sk_buff {
     /* These two members must be first. */
     struct sk_buff      *next;
     struct sk_buff      *prev;

     struct net_device   *dev;

     struct sock         *sk;

     ktime_t        tstamp;

     /* This is the control buffer. It is free to use for every layer.  */
     char           cb[48] __aligned(8);
     ...
     ...
```

Socket buffer

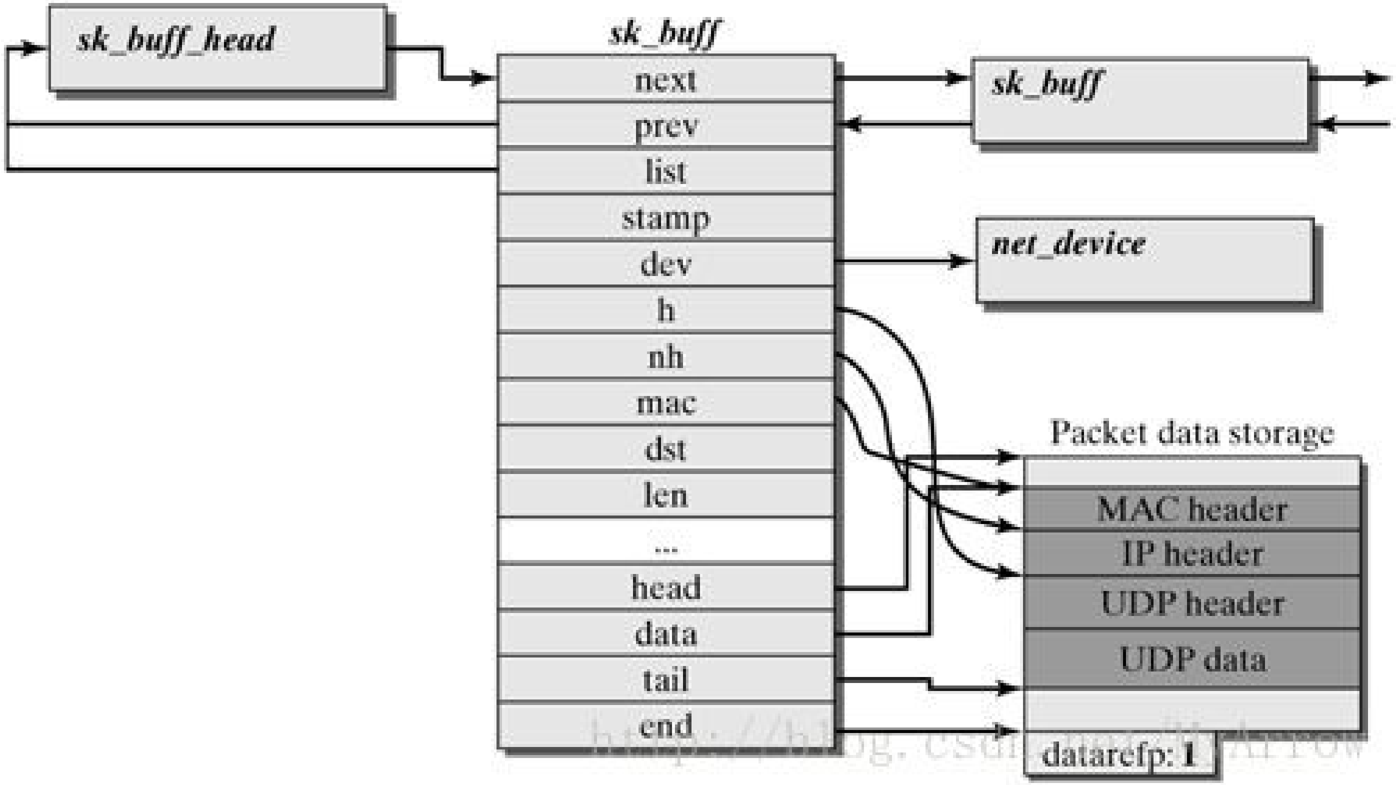
```
    ...

    unsigned int        len,
                        data_len;
    __u16               mac_len,
                        hdr_len;
    ...
    __u8                cloned:1,
    ...

    /* These elements must be at the end, see alloc_skb() for details.  */
    sk_buff_data_t      tail;
    sk_buff_data_t      end;
    unsigned char       *head,
                        *data;
    unsigned int        truesize;
    refcount_t          users;
}; /* struct sk_buff */
```

Socket buffer

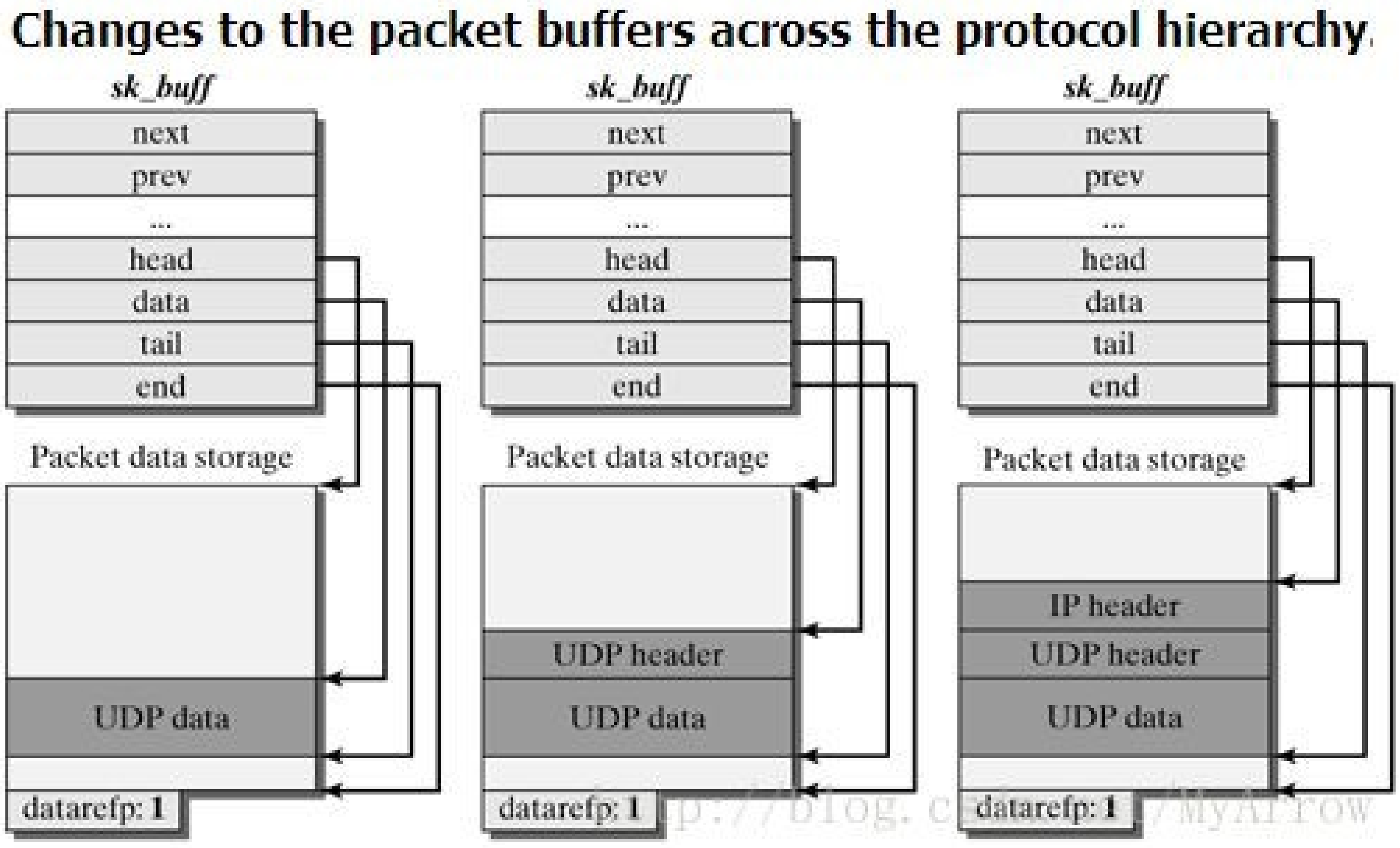
```
/* This data is invariant across clones and lives at
 * the end of the header data, ie. at skb->end.
 */

struct skb_shared_info {
	...
	__u8		nr_frags;
	__u8		tx_flags;
	...
	struct sk_buff *frag_list;
	...

	atomic_t	dataref;

	...

	skb_frag_t	frags[MAX_SKB_FRAGS];
};
```

```
struct skb_frag_struct {
	struct {
		struct page *p;
	} page;
	__u32 page_offset;
	__u32 size;
};
```

Socket buffer



Socket buffer

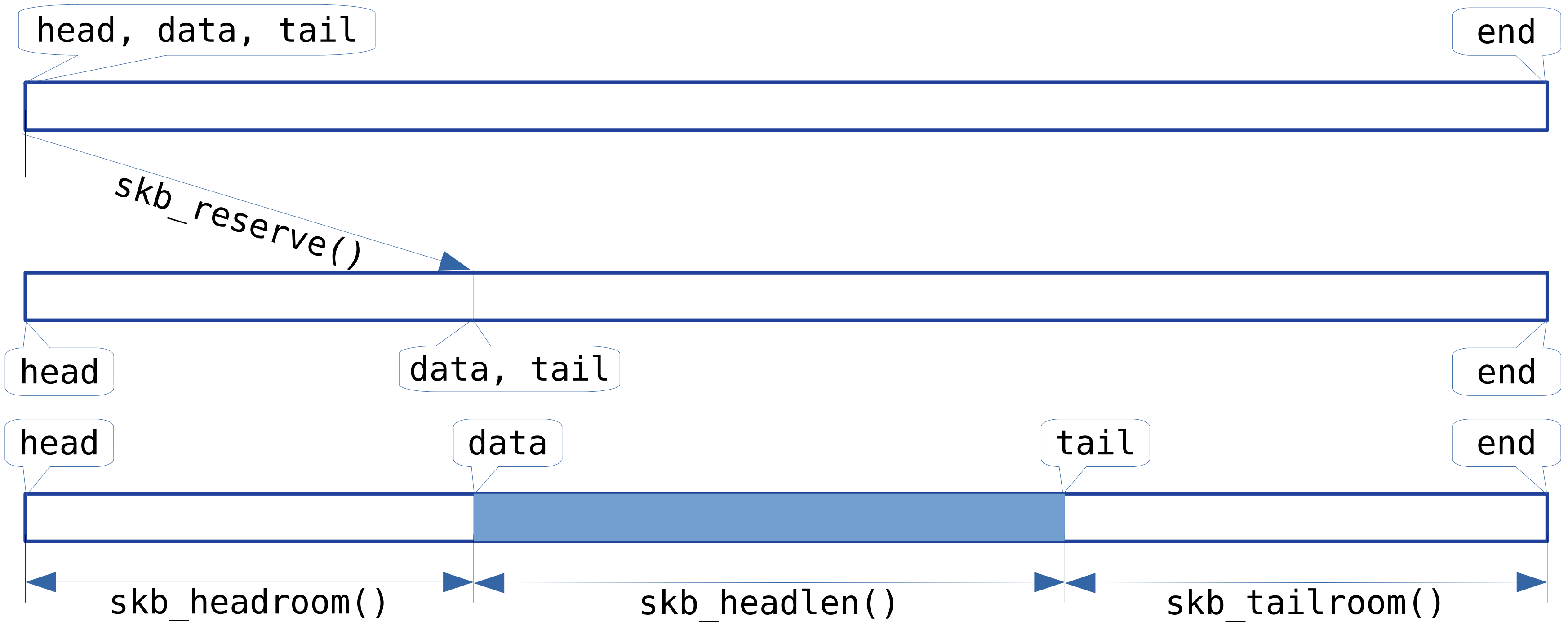


Socket buffer API

```
/* linux/skbuff.h */

struct sk_buff *alloc_skb(unsigned int size, gfp_t priority);

struct sk_buff *alloc_skb_with_frags(unsigned long header_len, unsigned long data_len,
                                     int max_page_order, int *errcode, gfp_t gfp_mask);

void kfree_skb(struct sk_buff *skb);
void kfree_skb_list(struct sk_buff *segs);

struct sk_buff *skb_copy(const struct sk_buff *skb, gfp_t priority);
struct sk_buff *skb_clone(struct sk_buff *skb, gfp_t priority);

bool skb_is_nonlinear(const struct sk_buff *skb);
int skb_linearize(struct sk_buff *skb);

struct sk_buff *skb_realloc_headroom(struct sk_buff *skb, unsigned int headroom);
```

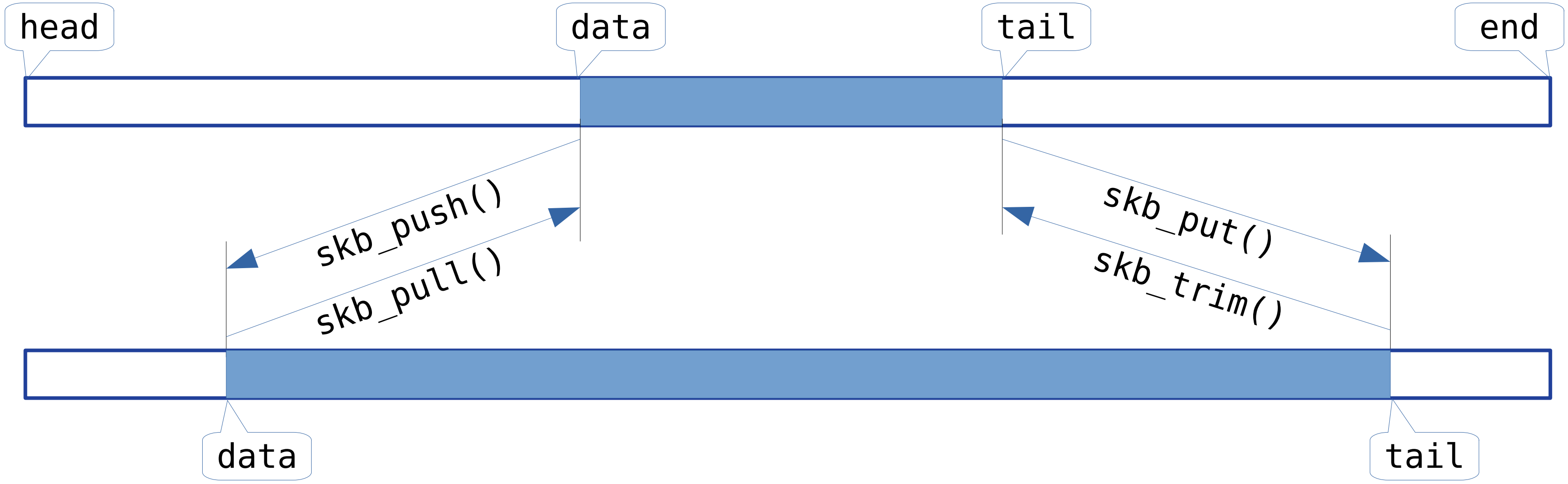
Socket buffer API

```
void skb_reserve(struct sk_buff *skb, int len);
unsigned int skb_headroom(const struct sk_buff *skb);
unsigned int skb_tailroom(const struct sk_buff *skb);
unsigned int skb_headlen(const struct sk_buff *skb);
```



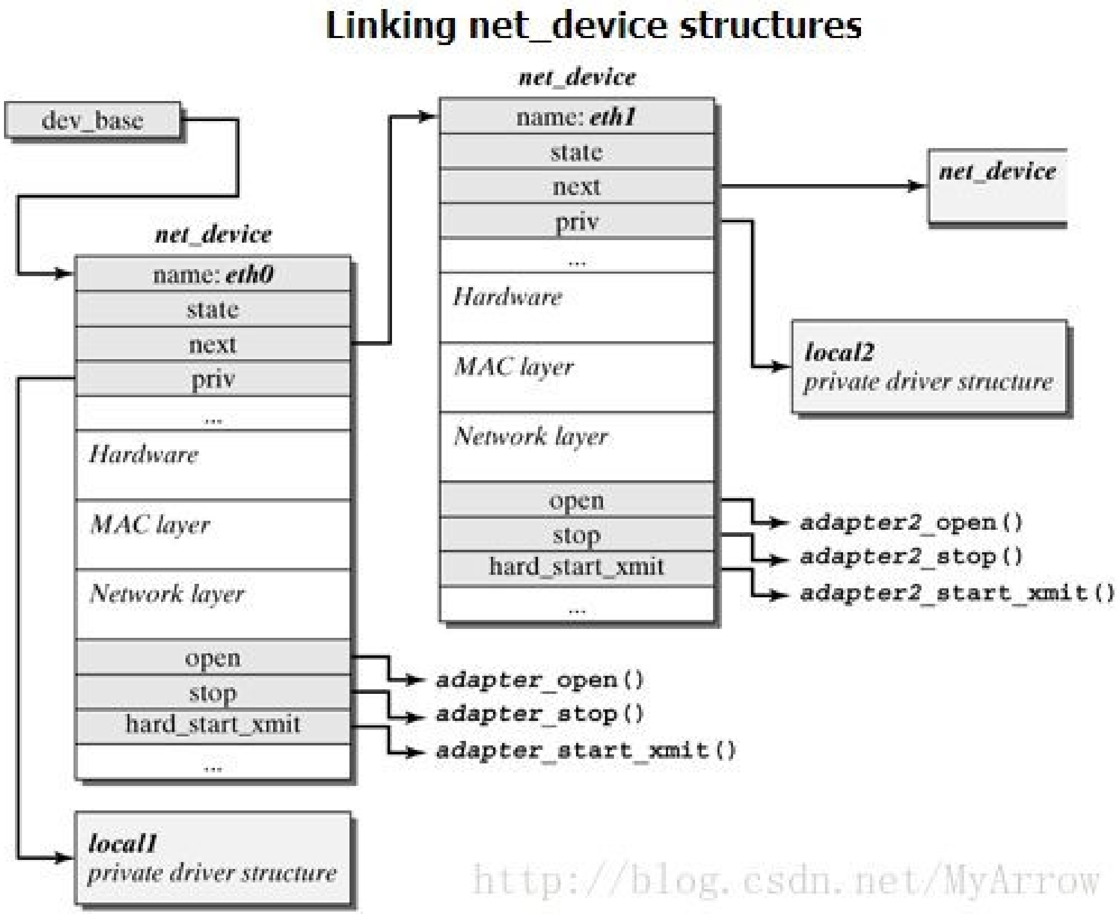
Socket buffer API

```
/* downward */
void *skb_push(struct sk_buff *skb, unsigned int len);
void *skb_put(struct sk_buff *skb, unsigned int len);

/* upward */
void *skb_pull(struct sk_buff *skb, unsigned int len);
void skb_trim(struct sk_buff *skb, unsigned int len);
```



Net device

```
/* linux/netdevice.h */

struct net_device;
```



OAPI vs NAPI

OAPI

Because packets are received in the interrupt context, the handler routine may perform only essential tasks so that the system (or the current CPU) is not delayed in performing its other activities for too long. In the interrupt context, data are processed by three short functions that carry out the following tasks:

- Interrupt handler determines whether the interrupt was really raised by an incoming packet (other possibilities are, e.g., signaling of an error or confirmation of a transmission as performed by some adapters).
- The packet contents are then transferred from the network card into the buffer and therefore into RAM, where the header data are analyzed using library functions available in the kernel sources for each transmission type. This analysis determines the network layer protocol used by the packet data—IP, for instance.
- Then netif_rx is called. Its call marks the transition between the card-specific part and the universal interface of the network layer. The purpose of this function is to place the received packet on a CPU-specific wait queue and to exit the interrupt context so that the CPU can perform other activities.

The kernel manages the wait queues of incoming and outgoing packets in the globally defined softnet_data array , which contains entries of type softnet_data. To boost performance on multiprocessor systems, wait queues are created per CPU to support parallel processing of packets. Explicit locking to protect the wait queues against concurrent access is not necessary because each CPU modifies only its own queue and cannot therefore interfere with the work of the other CPUs.
input_pkt_queue uses the sk_buff_head list head mentioned above to build a linked list of all incoming packets.
netif_rx marks the soft interrupt NET_RX_SOFTIRQ for execution before it finishes its work and exits the interrupt context.
net_rx_action is used as the handler function of the softIRQ.

OAPI vs NAPI

NAPI

Each time a frame arrives, an IRQ is used to signalize this to the kernel. This implies a notion of ‘‘fast’’ and ‘‘slow.’’ For slow devices, servicing the IRQ is usually finished before the next packet arrives . Since the next packet is also signaled by an IRQ, failing to fulfill this condition — as is often the case for ‘‘fast’’ devices — leads to problems . Modern Ethernet network cards operate at speeds of 10,000 MBit/s, and this would cause true interrupt storms if the old methods were used to drive them.
However if a new IRQ is received while packets are still waiting to be processed , no new information is conveyed to the kernel: It was known before that packets are waiting to be processed, and it is known afterward that packets are supposed to be processed — which is not really any news. To solve this problem, NAPI uses a combination of IRQs and polling .
Assume that no packets have arrived on a network adapter yet, but start to come in at high frequency now.
This is what happens with NAPI devices:

- The first packet causes the network adapter to issue an IRQ . To prevent further packets from causing more IRQs, the driver turns off Rx IRQs for the adapter. Additionally, the adapter is placed on a poll list.
- The kernel then polls the device on the poll list as long as no further packets wait to be processed on the adapter. Push the packets for the processing into the upper layers using netif_receive_skb function.
- Rx interrupts are re-enabled again.

If new packets arrive while old packets are still waiting to be processed, the work is not slowed down by additional interrupts .
While polling is usually a very bad technique for a device driver (and for kernel code in general), it does not have any drawbacks here: Polling is stopped when no packets need to be processed anymore , and the device returns to the normal IRQ mode of operation. No unnecessary time is wasted with polling empty receive queues as would be the case if polling without support by interrupts were used all the time.

OAPI vs NAPI

NAPI (cont.)

The NAPI method can only be implemented if the device fulfills two conditions :

- The device must be able to preserve multiple received packets, for instance, in a DMA ring buffer.
- It must be possible to disable IRQs for packet reception. However, sending packets and other management functions that possibly also operate via IRQs must remain enabled.

What happens if more than one device is present on the system? This is accounted for by a round robin method employed to poll the devices.
Each NAPI device is placed on a poll list when the initial packet arrives into an empty Rx buffer. As is the very nature of a list, the poll list can also contain more than one device.
The kernel handles all devices on the list in a round robin fashion: One device is polled after another, and when a certain amount of time has elapsed in processing one device, the next device is selected and processed. Additionally, each device carries a relative weight that denotes the importance in contrast to other devices on the poll list. Large weights are used for faster devices, while slower devices get lower weights. Since the weight specifies how many packets are processed in one polling round, this ensures that faster devices receive more attention than slower ones. The key change in contrast to the old API is that a network device that supports NAPI must provide a poll function . The device-specific method is specified when the network card is registered with netif_napi_add . Calling this function also indicates that the devices can and must be handled with the new methods.

References

- LDD3, Chapter 17: Network Drivers
  - Example for LDD3 chapter 17.
- Original eth_dummy example
- Kernel sources:
  - drivers/net/dummy.c
  - drivers/net/loopback.c
  - drivers/net/ethernet/intel/e100.c
  - include/linux/ethtool.h
  - include/uapi/linux/ethtool.h