Deep Dive of INTEL E1000's Offload Features

Author: Chidambaram Ramanathan, COMPAS lab, STONY BROOK UNIVERSITY

Email: cramanathan@cs.stonybrook.edu

Table of Contents

Smart NIC

Speed of networking devices have improved tremendously over the past few years. The ethernet technology has grown to “Gigabit Ethernet” and then to “10 Gigabit Ethernet” and even “40 Gigabit Ethernet” are available in the market. The host CPU has to keep up with this increasing speed and has to handle packets at such a fast rate. It thus needs parallel support from the network device. As a result of this, the networking devices (viz, NIC) are becoming smarter and trying to take some burden off the host CPU.
Following are the features adopted by modern NIC's.

TCP Offload Engine (TOE)

TCP is a very complex protocol which was designed for slow and unreliable networks. The growth of the Internet in terms of “transmission speeds” has burdened each processor with enormous workload in processing a network packet. Processing a single TCP packet involves mechanisms like “connection establishment”, “acknowledgment”, “checksum calculation”, “sliding window”, “congestion control” and “connection termination”. If each packet is arriving at very high speed, then it becomes increasingly complex for the host CPU to manage the network as well the performing the users requests.

To alleviate this, TCP Offload Engine was introduced which offloads the entire TCP/IP stack to the network controller. TOE allows the OS to offload all TCP/IP traffic to specialized hardware on the network adapter. By relieving the host processor off this bottleneck, TOE can boost the performance of the device as a whole.

In addition to reducing the burden on this host processor, it also reduces PCI traffic. PCI becomes less efficient if small bursts of data are to be transferred from host memory to the NICs. When it comes to the TCP protocol, the number of small packets are higher (eg, acknowledgments). If these are typically generated on the host CPU and transmitted across the PCI bus and out the network physical interface, this impacts the host computer IO throughput. TOE comes to rescue by processing all of the TCP packets in the network controller itself. This avoids small bursts of PCI traffic.

TOE finds it complete use in IP storage protocols such iSCSI.

Large Segmentation Offload

TCP Segmentation Offload

When Large Segmentation Offload is applied to TCP it becomes TCP Segmentation Offload.

When a computer wants to send large chunks of data out over a network, the chunks should be broken down into smaller segments that can pass through intermediate network elements adhering to the network's MTU. This process is referred to as segmentation. Often, the TCP protocol in the host computer performs this segmentation. Offloading this work to NIC is called TCP Segmentation Offload. It reduces the overhead of host CPU on fast networks.

Use of TSO can boost networking performance considerably. When one is dealing with thousands of packets every second, even a slight per-packet assist will add up. TSO reduces the amount of work needed to build headers and checksum the data, and it cuts down on the number of times that the driver must program operations into the network adapter.

This feature relies on the NIC to segment the data and then add the TCP, IP and Data-Link headers to each segment.

Generic Segmentation Offload

When the segmentation is applied to many protocols, not only TCP, it is called Generic Segmentation Offload. It is just a generalization of the concept of TSO.

Large Receive Offload

Just like TSO is for transmit side, LRO is for receive side. In short, LRO assists the host in processing incoming packets by aggregating them on-the-fly intro fewer but larger packets.

LRO combines received TCP packets to a single larger TCP packet and passes them then to the network stack in order to increase performance (throughput). This reduces the number of times, the device has to communicate with the driver. This again reduces the PCI traffic. More the packets that can be aggregated, higher the performance would be.

Checksum Offload

This is a feature with which the IP, TCP and UDP checksums are calculated on the NIC just before they are transmitted on the wire. This is particularly useful for TCP packets, as the checksum has to calculated for the data. When packets are to be sent and received at line rate, it would be a big advantage if the host CPU asked the device to calculate the checksum for the individual packets. Similarly on the receive side, when the host gets the packet with the checksum already calculated, then the protocols can directly start acting on the packet instead of having it checksummed before making decisions on it. In case of IP, the performance benefit is very negligible since IP needs checksum to be calculated only for the headers. It is done both in transmit and receive side.

Receive Side Scaling

RSS enables network adapters to distribute the kernel-mode network processing load across multiple processor cores in multi-core computers. The distribution of this processing makes it possible to support higher network traffic loads than would be possible if only a single core were to be used.

Contemporary NICs support multiple receive and transmit descriptor queues (multi-queue). On reception, a NIC can send different packets to different queues to distribute processing among CPUs. The NIC distributes packets by applying a filter to each packet that assigns it to one of a small number of logical flows. Packets for each flow are steered to a separate receive queue, which in turn can be processed by separate CPUs. This mechanism is generally known as “Receive-side Scaling” (RSS).

Scatter-Gather IO

Network packets need not be of fixed size. Some packets can be larger than the size of the data structure used to represent a packet. In that case, we might completely fill one packet and allocate another instance of the data structure to hold the rest of the data. In this case, a lot of memory may be wasted. Instead we can allocate page fragments which is only as big as the size of the excess data. When such fragments are allocate to store the contents, then doing a DMA to the device memory becomes difficult because the data is scattered. Scatter-gather I/O is a technique which handles the aforementioned problem. We pass a number of small buffers which make up a larger logical buffer. This is possible only if the device supports the scatter-gather technique. We can know whether a device supports a feature or not, during the initialization phase of the device.

We can thus see that, modern networking devices are becoming increasingly smart and contribute more towards increasing the performance of the host system. We will analyze in detail about few of the features which are supported in the INTEL E1000's driver. They are TX/RX Checksumming, TCP Segmentation Offload and Generic Receive Offload.

The document first tries to explain the infrastructure given by the stable version of the latest Linux Kernel version 3.12. It then delves into some of the basic data structures involved in Linux wrto networking. The last part in explaining the infrastructure tells about the device and driver communication and the format they follow.

Once the basics are touched upon, the document explains in dept with source code snippets as to how the TX/RX Checksumming and TSO/GRO work.

The document finally concludes by talking about Net-FPGA's NF NIC.

Packet Transmission

The following are the overview of the steps involved in transmitting a packet with INTEL E1000.

1. The protocol stack receives from an application a block of data that is to be transmitted.
2. The protocol stack calculates the number of packets required to transmit this block based on the MTU size of the media and required packet headers.
3. For each packet of the data block:
    1. Ethernet, IP and TCP/UDP headers are prepared by the stack.
    2. The stack interfaces with the software device driver and commands the driver to send the individual packet.
    3. The driver gets the frame and interfaces with the hardware.
    4. The hardware reads the packet from host memory via DMA transfers and transmits only after it has completely fetched all packet data and deposited into the on-chip transmit FIFO.

This seemingly simple sequence of steps has got a lot of things working behind in-order to achieve the transmission. We will see in great depth of how things get rolling. We are considering INTEL E1000 family as an example and try to understand the events. The happenings are analogous to that of our NetFPGA's 10G NF NIC.

Device Registration

When the Linux kernel boots, all PCI buses and devices are scanned and some kernel data structures are created to map these devices. A data structure is created to map each discovered PCI device on the PCI bus. The PCI bus structure *(struct pci_bus)* is created for a PCI bus and a pci_dev structure *(struct pci_dev)* is created for each PCI device. Each PCI device is uniquely identified by the structure *(struct pci_device_id)* which contains fields that uniquely identifies the device.

After this, the kernel creates a linked lists of all the PCI devices that has been identified and does a matching when the PCI driver for each device has been loaded.

The remaining step is that, the PCI driver for the device has to be loaded. It can happen in two ways. Either it is already built into the kernel or it is a loadable module which we have to do an *insmod* to load the driver.

When the driver is loading, the 1st function to be called is the init function. This does the important role of registering with the PCI layer. This is summarized as follows

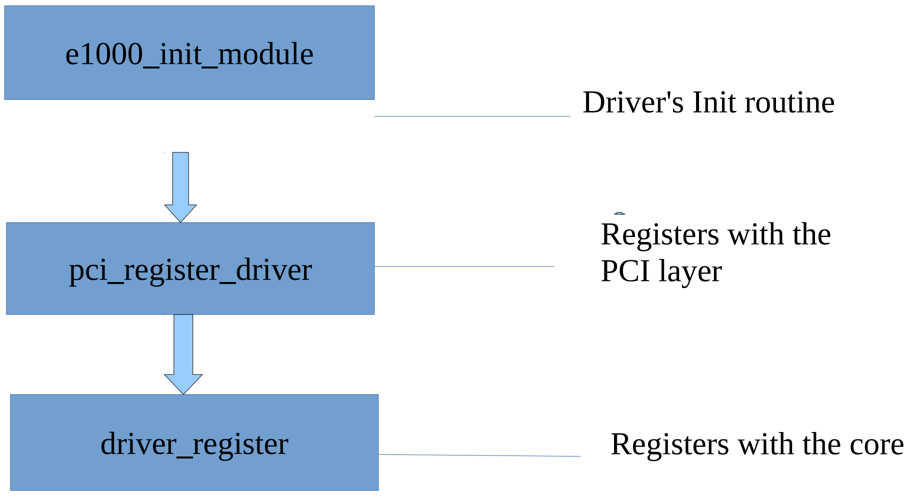


Figure summarizing the registration process of a device such as a NIC

The important thing to be noted here is that, each driver will have a driver structure. In the case of e1000, the structure is *(e1000_driver)*. It contains critical information used by the PCI layer to match the list of identified devices. It also supplies the address of the probe function *(e1000_probe)* which is another important function in the context of registering a device. The probe function will be called by the PCI layer and that initializes the rest of the driver part. The probe function also announces the device's presence to the Linux Kernel.

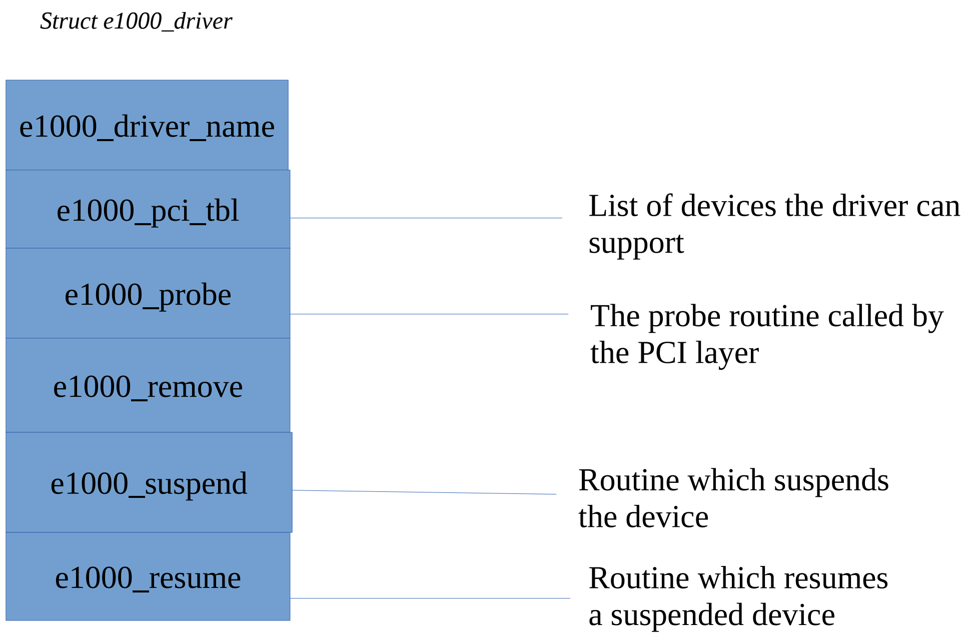

Struct e1000_driver
e1000_driver_name
e1000_pci_tbl
e1000_probe
e1000_remove
e1000_suspend
e1000_resume
List of devices the driver can support
The probe routine called by the PCI layer
Routine which suspends the device
Routine which resumes a suspended device

Figure showing the structure of the e1000's driver

In the above structure, other than the probe routine, the *e1000_pci_tbl* value is important. It tells to the PCI layer what devices it can drive. So if the driver can manage more than one device, it contains the ID's of all the devices it can manage. The PCI layer will associate this driver with the all the devices that the driver mentioned it can drive.

Hardware – Software Interface

Our device has now been registered with the kernel and is set to transmit packets. Let us try to understand how the software interfaces with the hardware to transmit data. There is a separate ring structure used for transmission and another ring structure used for reception of packets. The structure contains several descriptors each of which will be associated with a buffer. The buffer stores the actual contents of the packets and the corresponding descriptor contains the address of this buffer. The stack gives the packets it wants to transmit to the driver. The driver puts the data into the buffer and updates the descriptor and indicates the device that there is a new packet to transmit. The device will now copy these into its FIFO buffer and starts transmitting the packets. Once the packets has been transmitted, it updates a specific field in the descriptor and indicates the software that a packet has been successfully sent out of the wire. Lets see the structure of the ring, buffers and descriptors in the following sections.

DMA Engine and FIFO

The DMA engine handles the receive and transmit of data and descriptor transfers between the host memory and the on-chip memory. In the receive path, the DMA engine transfers the data stored in the receive data FIFO buffer to the receive buffer in the host memory, specified by the address in the descriptor. It also fetches and writes back updated receive descriptors to host memory. In the transmit path, the DMA engine transfers data stored in the host memory buffers to the transmit data FIFO buffer. It also fetches and writes back updated transmit descriptors.

Buffer and Descriptor Structure

Software allocates the transmit and receive buffers and also forms the descriptors that contain the pointers to and the status of those buffers. A conceptual ownership exists between the driver software and the hardware of the buffers and descriptors. The software gives the hardware ownership of a queue of buffers for receives. The receive buffers store data that the software then owns once a valid packet arrives.

For transmitting a packet, the software owns a buffer until it is ready to transmit. The software then commits the buffer to the hardware. The hardware then owns the buffer until the data is loaded or transmitted in the transmit FIFO.

Descriptors store the following information about the buffers.

1. Physical address
2. Length
3. Status and command information about the referenced buffer.
4. End of packet field that indicates the last buffer for a packet.
5. Type of packet

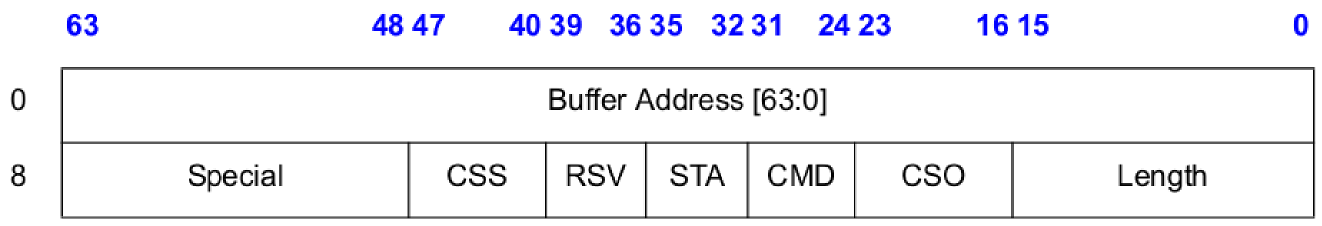
6. Specific operations to perform in the context of transmitting a packet such as checksum offloading etc.

Transmit Descriptor Structure

The Ethernet Controller provides three types of transmit descriptor formats.

Legacy Descriptor

This is the original descriptor.



Buffer Address - Address of the associated buffer in the host memory.

Length - Length of the packet

CSO – Checksum Offset field indicates where, relative to start of the packet to insert a TCP checksum.

CMD – Command field

STA – Status field.

CSS – Checksum start field indicates where to begin computing the checksum.

Although, the hardware can be programmed to calculate and insert TCP checksum using legacy descriptor format, it is better to use the newer TCP/IP Context transmit Descriptor format. This newer format allows the hardware to calculate both the IP and TCP checksums for outgoing packets.

TCP/IP Context Transmit Descriptor Format

This provides access to the enhanced checksum offload facility. This feature allows TCP and UDP packet types to be handled by performing additional work in the hardware and thus reducing the software overhead with preparing these packets for transmission.

This does not point to the packet data as a data descriptor does. Instead, this descriptor provides access to an on-chip context that supports the transmit checksum offloading feature on the controller.

The context is explicit and directly accessible via the TCP/IP context transmit descriptor. The context is used to control the checksum offloading feature for normal packet trasmission. A context can be used for multiple packets unless a new context is loaded prior to each new packet. For example if most of the traffic generated from a given node is standard TCP frames, this context could be setup once and used for many frames. This avoids the extra descriptor penalty for each packet.

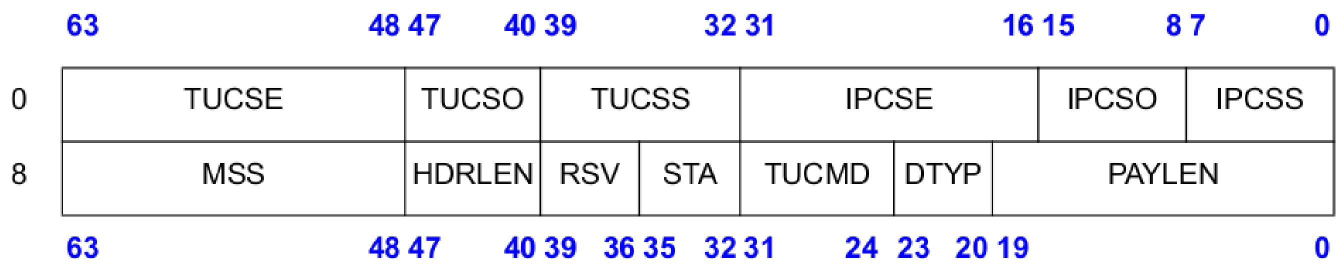
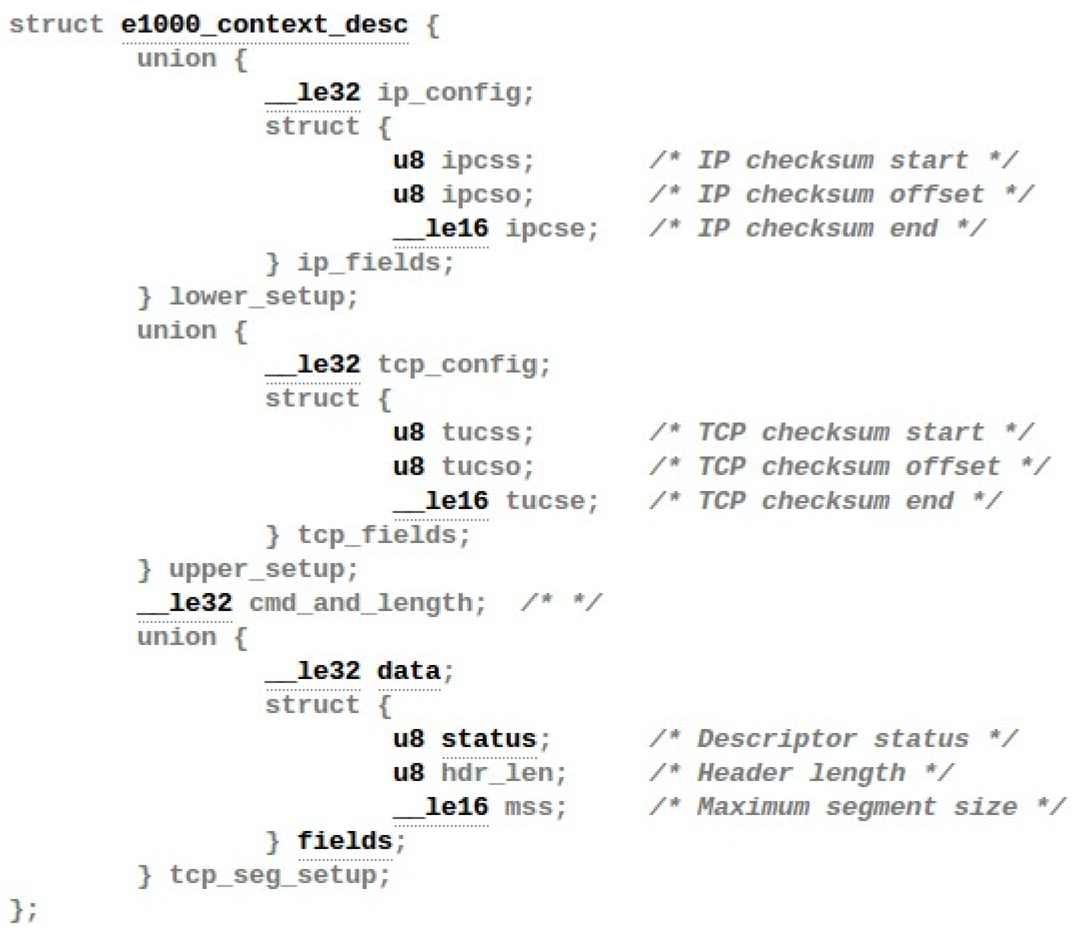


Figure representing the structure of the context descriptor



```
struct e1000_context_desc {
        union {
                __le32 ip_config;
                struct {
                        u8 ipcss;       /* IP checksum start */
                        u8 ipcso;       /* IP checksum offset */
                        __le16 ipcse;   /* IP checksum end */
                } ip_fields;
        } lower_setup;
        union {
                __le32 tcp_config;
                struct {
                        u8 tucss;       /* TCP checksum start */
                        u8 tucso;       /* TCP checksum offset */
                        __le16 tucse;   /* TCP checksum end */
                } tcp_fields;
        } upper_setup;
        __le32 cmd_and_length;  /* */
        union {
                __le32 data;
                struct {
                        u8 status;      /* Descriptor status */
                        u8 hdr_len;     /* Header length */
                        __le16 mss;     /* Maximum segment size */
                } fields;
        } tcp_seg_setup;
};
```

Figure representing the structure associated with the context descriptor

TCP/IP Data Descriptor Format

This is similar to the legacy mode descriptor but also integrates the checksum offloading and the TCP segmentation feature.

It is the companion to the TCP/IP context transmit descriptor. It has four important fields. The command field, the status field, option field and the special field. Out of this, the option field has information related to TCP/UDP checksum. The two important fields in the option field are TXSM and the IXSM field. When TXSM is set, the value in TCP/UDP field is modified by the hardware. Similarly, when the IXSM field is set, the IP checksum is modified by the hardware. Both the fields are valid only in the first data descriptor for a given packet or TCP Segmentation context.

It also has fields which tell the software that a packet has been successfully sent on the wire.

Receive Descriptor Structure

Upon receipt of a packet for Ethernet controllers, hardware stores the packet data into the indicated buffer and writes the length, Packet Checksum, status, errors, and status fields. Length covers the data written to a receive buffer including CRC bytes. Software must read multiple descriptors to determine the complete length for packets that span multiple receive buffers.

Receive Descriptor

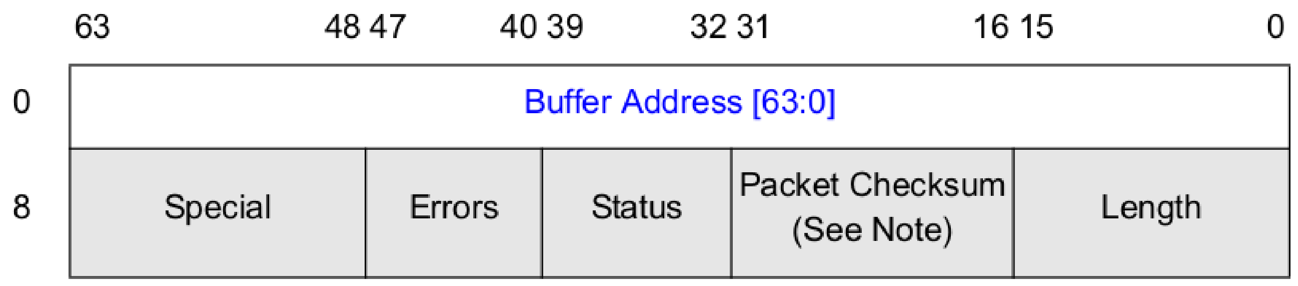


Figure Illustrating the structure of a receive descriptor in INTEL E100

The above figure shows the format of a receive descriptor. The Buffer Address field contains the physical address of the socket buffer to which the device can DMA the data to. Out of the other fields, the status field is of interest here.

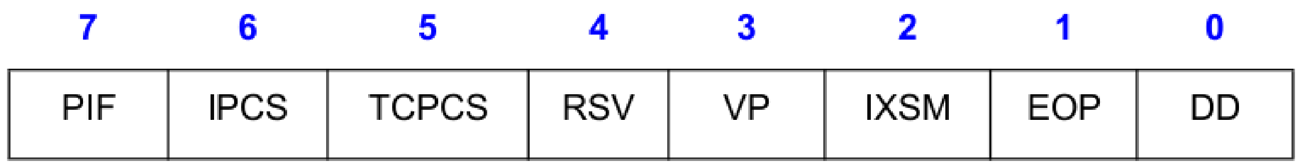
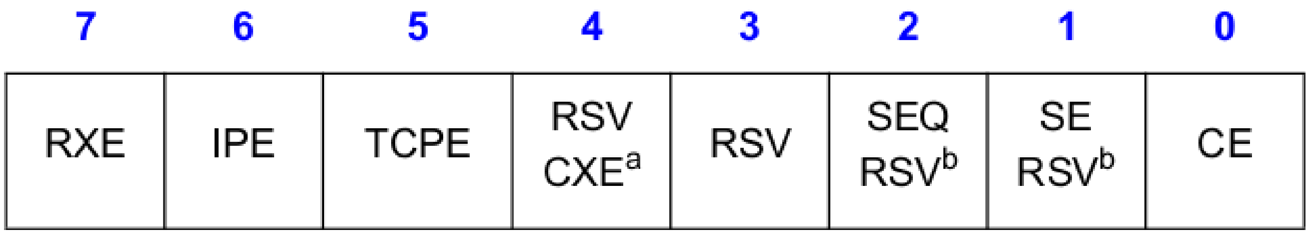


Figure which shows the contents of the status field in a receive descriptor

ICPS – This field indicates whether the hardware has performed checksum on the input packet. When this bit is 0, it means that the hardware has already performed the checksum on the input packet. If the bit is 1, it indicates the stack to calculate the checksum.

TCPCS – This is analogous to the above field. It tells whether the device has calculated the TCP Checksum. If the bit is 0, it means that the device has already calculated and if the bit is 1, it means that the stack has to calculate the checksum for this input packet.

IXSM – This field indicated whether the value in the above filed should be ignored or not. When the bit is 1, it means that the value should be ignored and when the bit is 0, then whatever is specified in the above fields holds good.



7	6	5	4	3	2	1	0
RXE	IPE	TCPE	RSV CXE[a]	RSV	SEQ RSV[b]	SE RSV[b]	CE

Figure Illustrating the contents of the error field in a receive descriptor

The device has to also indicate if the calculated checksum is valid or not. I.e It has to tell the driver if the checksum calculation passed or not. It does so, with the help of the error field in receive descriptor.

IPE – IP Checksum Error. If the bit is set, it indicates that the IP Checksum error is detected in the received packet.
TCPE – Similar to the above field. It is set, when TCP/UDP checksum error is detected in the received packet.

Transmit Descriptor Ring Structure

Now that we have seen the structure of the descriptors and how they are passed to the hardware, we will now see the structure of the ring buffer as a whole.

From hardware perspective, there are a pair of registers which maintains the transmit queue. To transmit more packets, the software writes descriptors to the ring and moves the ring's tail pointer. The tail pointer points one entry beyond the hardware owned descriptor. Transmission continues up to the descriptor where head equals tail at which point, the queue is empty.

Care should be taken by the software so that, the descriptors passed to the hardware should not be manipulated until the head pointer has advanced past them.

The transmit descriptor ring has the following registers

1. Transmit Descriptor Base Address Registers (TDBAL / TDBAH) – High and low 32 bit address of start of the descriptor buffer ring.
2. Transmit Descriptor Length (TDLEN) – Number of bytes allocated to the circular buffer or length of the ring.
3. Transmit Descriptor Head Register (TDH) – Head register. Indicates the in-progress descriptor already loaded in the output FIFO.
4. Transmit Descriptor Tail Register (TDT) – Tail register. Offset from base indicating the location beyond last descriptor of hardware.

The hardware also pre-fetches the descriptors which are aligned on the cache boundary. Once a packet has been sent on the wire, it sets the DD bit in the descriptor to indicate that a packet has been successfully sent.

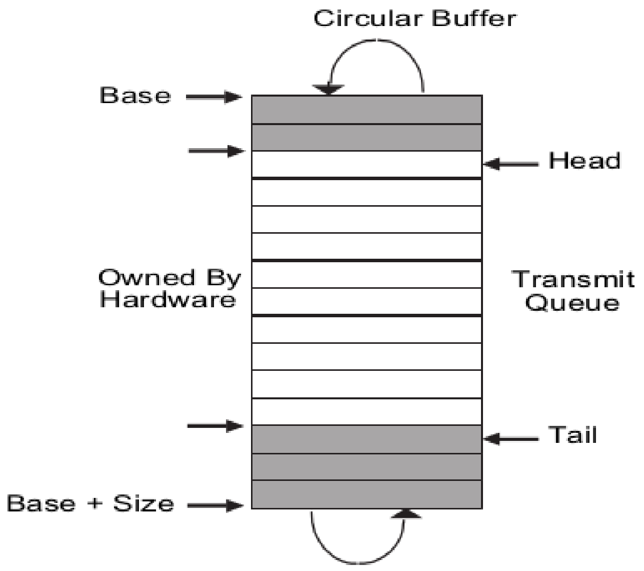
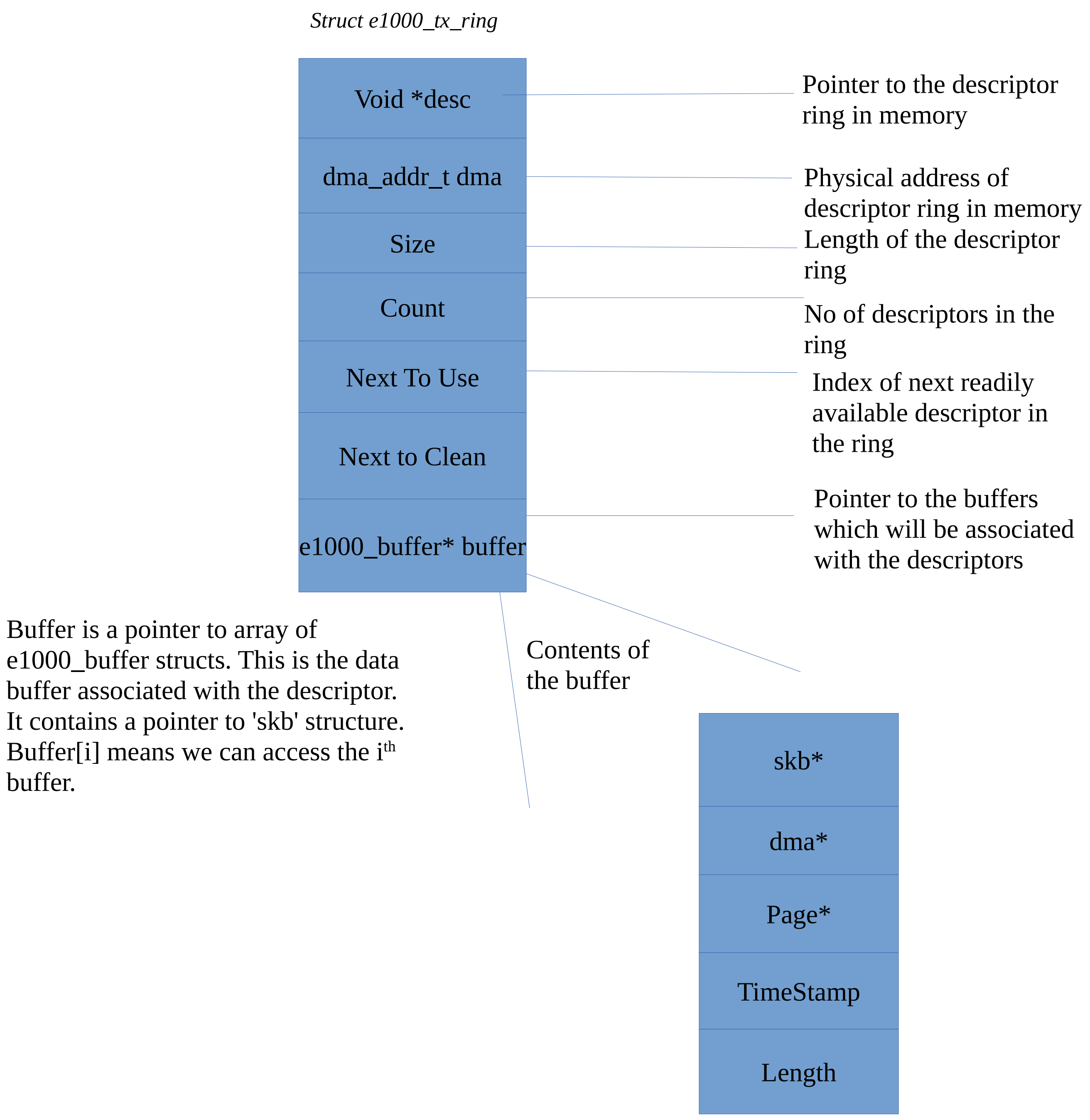


Figure depicting the TX RING which is used for transmission

Structure of TX Ring


Struct e1000_tx_ring
Void *desc
dma_addr_t dma
Size
Count
Next To Use
Next to Clean
e1000_buffer* buffer
Pointer to the descriptor ring in memory
Physical address of descriptor ring in memory
Length of the descriptor ring
No of descriptors in the ring
Index of next readily available descriptor in the ring
Pointer to the buffers which will be associated with the descriptors
Contents of the buffer
skb*
dma*
Page*
TimeStamp
Length

Buffer is a pointer to array of e1000_buffer structs. This is the data buffer associated with the descriptor. It contains a pointer to 'skb' structure. Buffer[i] means we can access the ith buffer.

Figure representing the base structure of INTEL E1000's TX_RING.

Allocating resources for TX Ring

The above structure ***e1000_tx_ring*** is created once when the driver is initializing the device. The exact definition of the structure can be found in ***drivers/net/ethernet/intel/e1000/e1000.h***. Memory for the tx_ring and the rx_ring are allocated during the function call ***e1000_alloc_queues.***

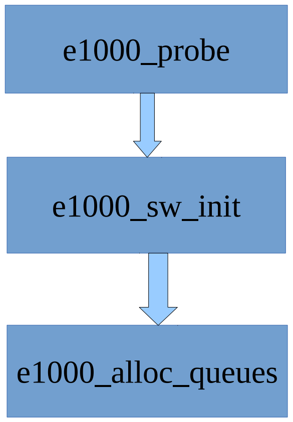


Figure explaining the sequence involved in allocating memory for the ring

Initializing resources for TX Ring

The fields of the tx_ring are setup while allocating the resources. It is precisely done in the function call ***e1000_setup_tx_resources.*** This method allocates the descriptors as well. The total count of the number of descriptors is mentioned by the driver. Based on this total count, it creates a memory of size 'S' which is equal to count * sizeof_each_descriptor. This size is aligned to page boundary.

*txdr->size = txdr->count * sizeof(struct e1000_tx_desc);*

Next step is to allocate the descriptors.

```
txdr->desc = dma_alloc_coherent(&pdev->dev, txdr->size, &txdr->dma, GFP_KERNEL);
```

The physical address of this descriptor in memory is stored in ***txdr->dma*** variable of the tx_ring. The virtual address of this ring is stored in ***txdr->desc.*** Thus variable 'desc' is now pointing to a valid descriptor ring in memory. The next task is to allocate memory for the buffers.

```
size = sizeof(struct e1000_buffer) * txdr->count;
txdr->buffer_info = vzalloc(size);
```

Thus, we can see that 'count' number of buffers are allocated to store the packet data. I.e 'count' number of descriptors and 'count' number of buffers are created. Once they set the value, they initialize the next usable descriptor as descriptor 0.

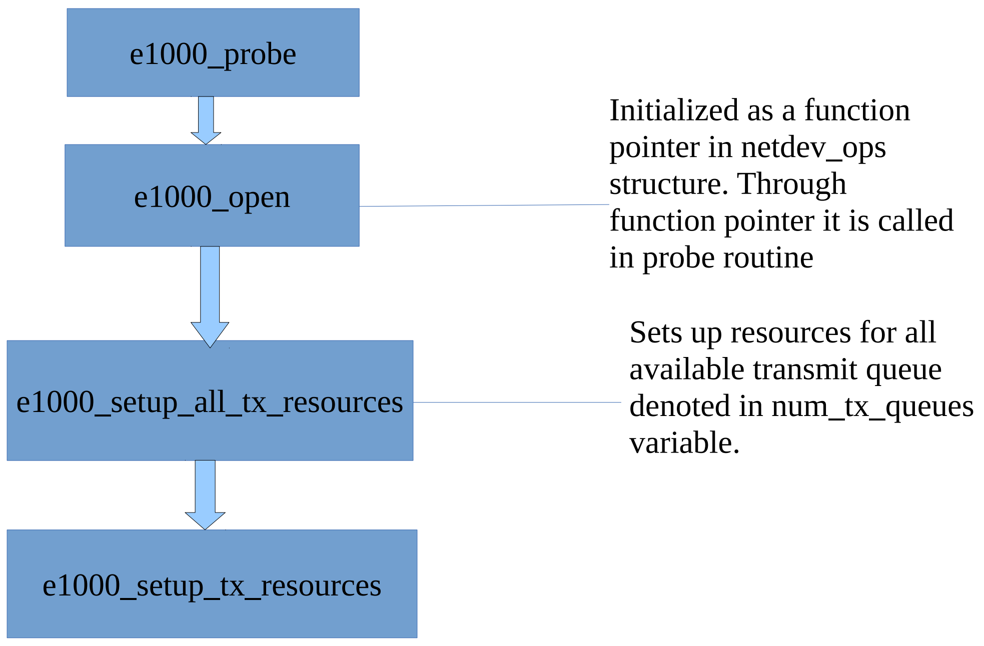

e1000_probe
e1000_open
e1000_setup_all_tx_resources
e1000_setup_tx_resources
Initialized as a function pointer in netdev_ops structure. Through function pointer it is called in probe routine
Sets up resources for all available transmit queue denoted in num_tx_queues variable.

Figure explaining the sequence in initializing the resources

Receive Descriptor Ring Structure

Software adds receive descriptors by writing the tail pointer with the index of the entry beyond the last valid descriptor. As packets arrive, they are stored in memory and the head pointer is incremented by hardware. The head points to the next descriptor that is written back. When the head pointer is equal to the tail pointer, the ring is empty.

Hardware stops storing packets in system memory until software advances the tail pointer, making more receive buffers available. Hardware owns all the descriptors between the head and the tail.

In other words, the NIC fetches descriptors till the TAIL pointer. Software writes to the tail pointer whenever it has data to be sent. Whenever the on-chip buffer is empty, the NIC fetches the descriptors till the tail pointer and stores in its cache line. To maximize memory efficiency, the receive descriptors are packed together and written as cache line whenever possible.

The receive descriptor ring has the following registers.

1. Receive Descriptor Base Address Register (RDBAL, RDBAH) – RDBAL contains the lower 32 bits and RDBAH contains the higher 32 bits. These registers indicate the start of the descriptor ring buffer.

2. Receive Descriptor Length (RDLEN) – This register determines the number of bytes allocated to the circular buffer.

3. Receive Descriptor Head (RDH) – This register holds a value that is an offset from the base and

indicates the in progress register. There can be upto 64K descriptors in the circular buffer.

4. Receive Descriptor Tail – This register holds a value that is an offset from the base, and identifies the location beyond the last descriptor the hardware can process.

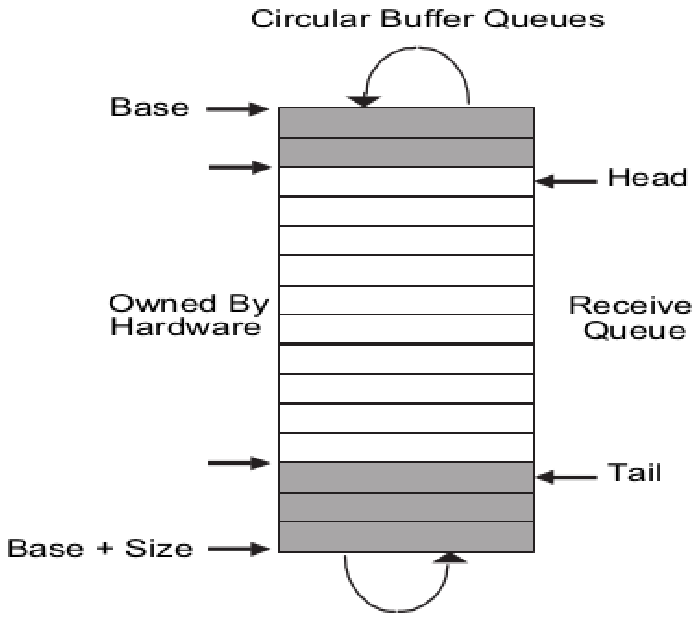
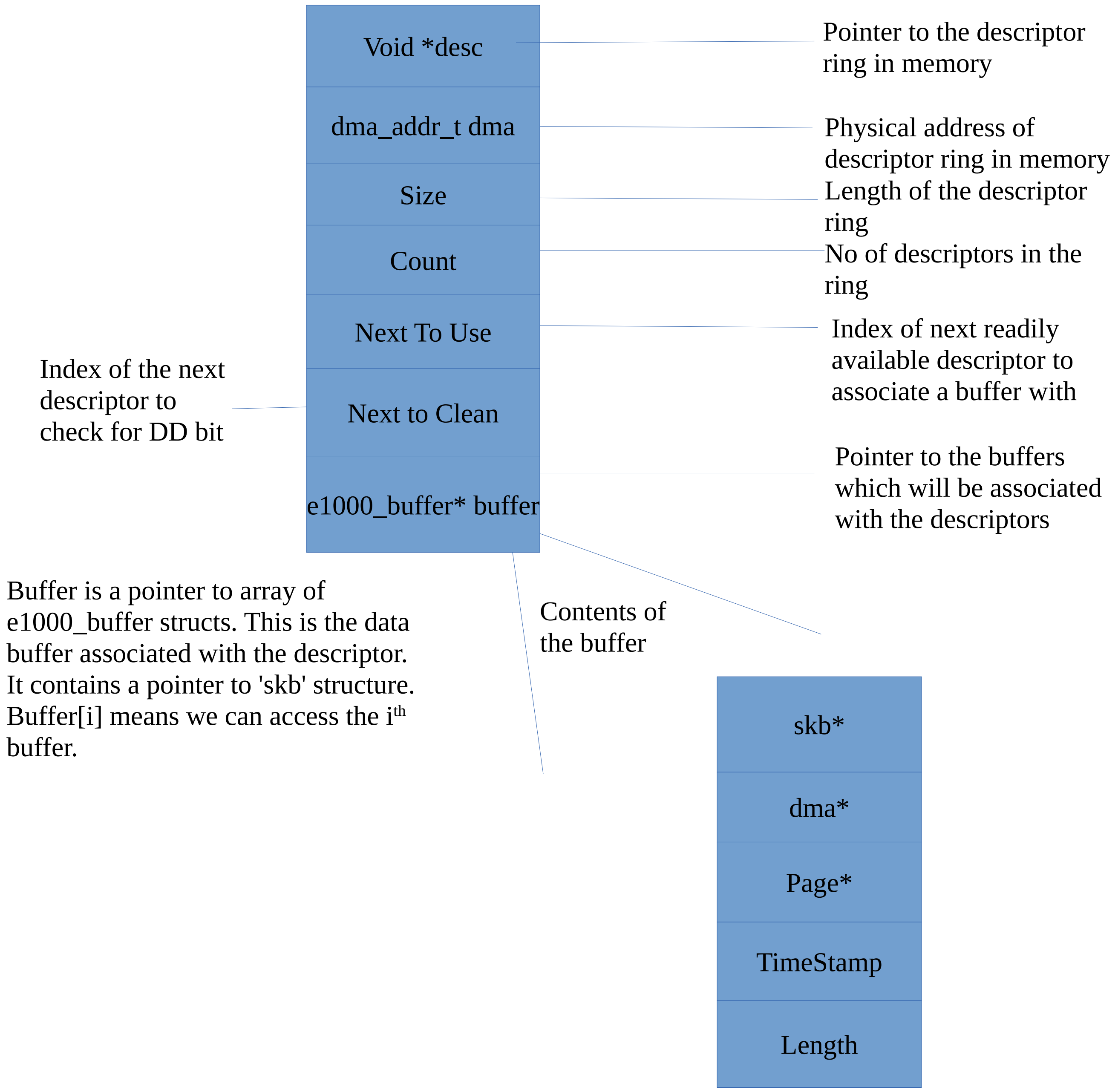


Figure Depicting the RX RING which is used in packet reception

Structure of RX Ring


Void *desc
dma_addr_t dma
Size
Count
Next To Use
Next to Clean
e1000_buffer* buffer
Pointer to the descriptor ring in memory
Physical address of descriptor ring in memory
Length of the descriptor ring
No of descriptors in the ring
Index of next readily available descriptor to associate a buffer with
Index of the next descriptor to check for DD bit
Pointer to the buffers which will be associated with the descriptors
Contents of the buffer
skb*
dma*
Page*
TimeStamp
Length

Buffer is a pointer to array of e1000_buffer structs. This is the data buffer associated with the descriptor. It contains a pointer to 'skb' structure. Buffer[i] means we can access the ith buffer.

Figure representing the base structure of INTEL E1000's TX_RING.

Allocating Resources for RX Ring

Resources for the rx ring is allocated in the same way as that of the tx ring.
The above structure ***e1000_rx_ring*** is created once when the driver is initializing the device. The exact definition of the structure can be found in ***drivers/net/ethernet/intel/e1000/e1000.h***. Memory for the tx_ring and the rx_ring are allocated during the function call ***e1000_alloc_queues.***

Initializing resources for RX Ring

Resources for rx rings are also allocated the same way as that of the tx ring. After allocating the resources, they are configured as follows. Depending on where the device can handle jumbo frames or not, the appropriate functions are called.

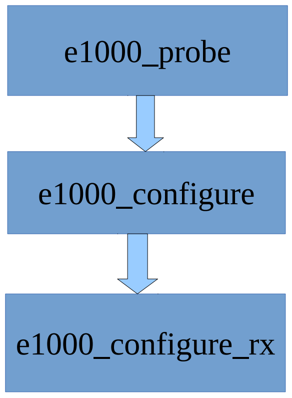
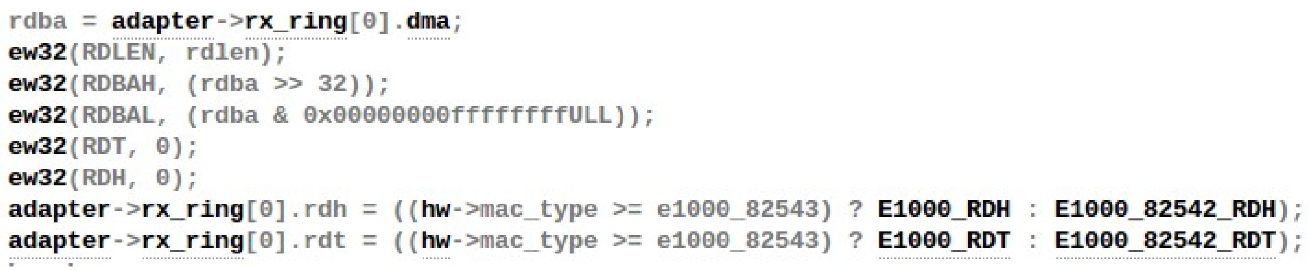


Figure Illustrating the functions involved in configuring the rx ring

The receive checksum is enabled by default for those devices which support it. This is also done in the configuration step. The registers mentioned above which are used to manipulate the rx ring structure are also initialized here.



```
rdba = adapter->rx_ring[0].dma;
ew32(RDLEN, rdlen);
ew32(RDBAH, (rdba >> 32));
ew32(RDBAL, (rdba & 0x00000000ffffffffULL));
ew32(RDT, 0);
ew32(RDH, 0);
adapter->rx_ring[0].rdh = ((hw->mac_type >= e1000_82543) ? E1000_RDH : E1000_82542_RDH);
adapter->rx_ring[0].rdt = ((hw->mac_type >= e1000_82543) ? E1000_RDT : E1000_82542_RDT);
```

Figure Illustrating the configuration of RX Ring

Calculation of Checksum by Device

The following is the convention used by the stack and the device driver to communicate about checksum calculation in the device.

1. NETIF_F_HW_CSUM – The NIC is a clever device and it can calculate all the checksums.
2. NETIF_F_IP_CSUM – The NIC can calculate checksum only TCP/UDP over IPV4
3. NETIF_F_IPV6_CSUM – Same as above, but in addition, IPV6.

There are two fields in the *skb* structure that is used by the driver to communicate with the L4 and vice versa.

1. skb->csum
2. skb->ip_summed

When a packet is received, *skb->csum* may hold the L4 checksum. *skb->ip_summed* field keeps track of the status of the L4 checksum. It can hold the following values

1. CHECKSUM_NONE – The checksum in *skb->csum* is not valid because
   1. Device does not support hardware checksumming
   2. Device computed the hardware checksum and found it to be corrupted.
2. CHECKSUM_HW – The NIC has computed the checksum on the L4 header and has copied it into the *skb->csum* field. The software only need to add the pseudo header to *skb->csum* and to verify the resulting checksum.
3. CHECKSUM_UNNECESSARY - NIC has computed and verified the checksum on the L4 header and checksum, as well as on the pseudo header. So the software is relieved from having to do any L4 checksum verification. Example is the loop-back device

When a packet is transmitted, *skb->ip_summed* can take the following values

1. CHECKSUM_NONE – The protocol has already taken care of the checksum and has copied it into the *skb->csum* field. Thus, there is no need to recompute the checksum.
2. CHECKSUM_PARTIAL – The protocol has stored into its header, the checksum on the pseudo header only. The device is supposed to complete it by adding the checksum on the L4 header.

While the feature flags NETIF_F_XXX_CSUM are initialized by the device driver when the NIC is enabled, the CHECKSUM_XXX flags have to be set for every *sk_buff* buffer that is received or transmitted.

At reception time, it is the device driver which initializes the *ip_summed* correctly based on the NETIF_F_XXX_CSUM capabilities.

At transmission time, the L3 transmission API's initialize ip_summed based on the checksumming capabilities of the egress device, which can be obtained from the routing table which includes information about the egress device and therefore its checksumming capabilities.

With the above setup, the driver has got a definite way to indicate the device that, it is the duty of the device to calculate the checksum. The driver has to first realize that the device has the capability of calculating checksum in it.

It identifies this capability of the device in the registration time. In the probe routine of the device, the driver checks the type of the device and updates the *features* field in the *net_device* structure. The applications running in the top of the stack might indicate that it needs the device to calculate the checksum. The transmit function in the kernel receives this input from the application and checks if the device has the support. If not, then it calls the *skb_checksum_help* function which calculates the checksum in the host processor itself. If the device supports checksum, it passes the control to the driver. The driver knows how to communicate with the device and instruct it to calculate the checksum.

Checksum calculation - Pseudo-header

The calculation of a checksum includes a pseudo header. A pseudo-header is pre-pended to the user datagram for the checksum computation. The pseudo header includes some fields from the IP header like the source IP address, destination IP header. The checksum in layer 4 header is calculated over the pseudo header, the layer 4 header and layer 4 payload. The pseudo header is not transmitted with the user datagram. It is reflected in the length field in the UDP header.
The main purpose of the pseudo header is to verify that the packet has reached the correct destination. The checksum calculated by sender covers destination IP address in the pseudo header. Receiver recreates the pseudo header using IP address from the header of the IP datagram that carried the UDP / TCP message. The receiver computes the checksum and compares it. But there is a problem with this method. It is about determining the destination IP address when calculating the checksum in the layer 4. Thus there is a layer violation using which the layer 4 interacts with layer 3 to get the source IP and destination IP address.
In short,

1. The Layer 4 software asks the IP to compute the source address and the destination address.
2. Constructs the pseudo header with the above information, layer 4 header and layer 4 payload.
3. Computes the checksum.
4. Discards the pseudo header
5. Passes the user datagram to IP for delivery.

It is to be noted that the software precalculates the partial pseudo header sum which includes IPv4 SA, DA and protocol types, but not the TCP length and stores this value into the TCP checksum field of the packet. The TCP length field is the TCP header length including option fields plus the data length in bytes, which is calculated by hardware on a frame by frame basis. The TCP length does not count the length of the pseduo header.

Detour on sk_buff

Before explaining the actual transmission it will be better, if the organization of the sk_buff is mentioned.

sk_buff in linux represents a packet. It consists of three segments.

1. sk_buff structure which is also referred to as a sk_buffer header.
2. Linear data block containing data
3. NonLinear data portion represented by struct skb_shared_info.

sk_buff contains linear and non-linear data. Linear data are represented by the data field of the sk_buff. Normally we allocate one page of linear data only for segments that can be accommodated in s a single page. In the case where the total segment length is more than one page, we have two options.

1. First is to have a linear data of length which can accommodate the entire segment.
2. Have a page data area for the rest of the packet.

The length of the sk_buff has certain fields which are confusing. They are *len* and *data_len*. The latter only comes into play when there is paged data in the SKB. skb->data_len tells how many bytes of paged data there are in the SKB. From this we can derive a few more things:

- The existence of paged data in an SKB is indicated by skb->data_len being non-zero. This is codified in the helper routine skb_is_nonlinear().
- *The amount of non-paged data at skb->data can be calculated as skb->len - skb->data_len.* There is a helper routine already defined for this called *skb_headlen().*
- *Len* – Total length of the segment.
- data_len – This field is used only when we have non-linear data associated with the sk_buff. This field indicates the portion of the total packet length that is contained as paged data, which means the linear data length will be *skb->len – skb->data_len.*

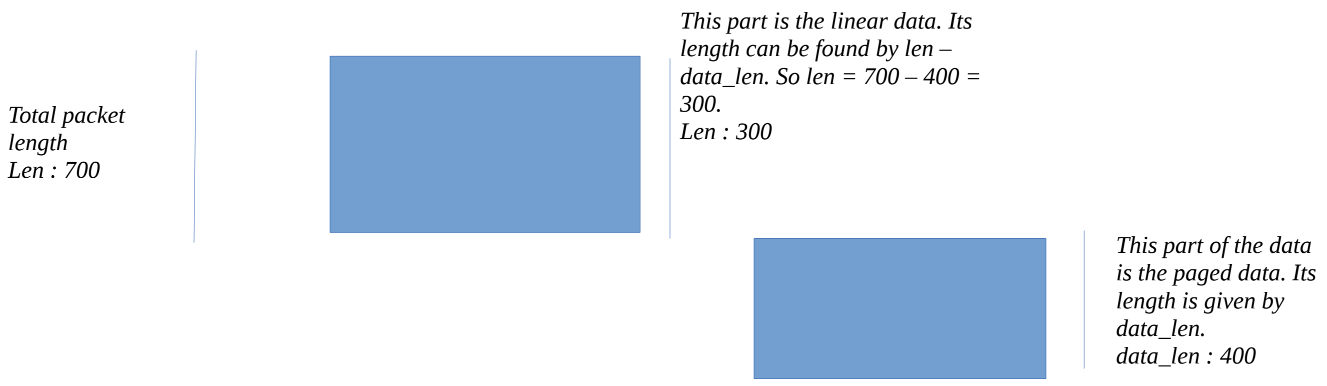

Total packet length
Len : 700
This part is the linear data. Its length can be found by len – data_len. So len = 700 – 400 = 300.
Len : 300
This part of the data is the paged data. Its length is given by data_len.
data_len : 400

Figure Illustrating the meaning of the length fields associated with sk_buff

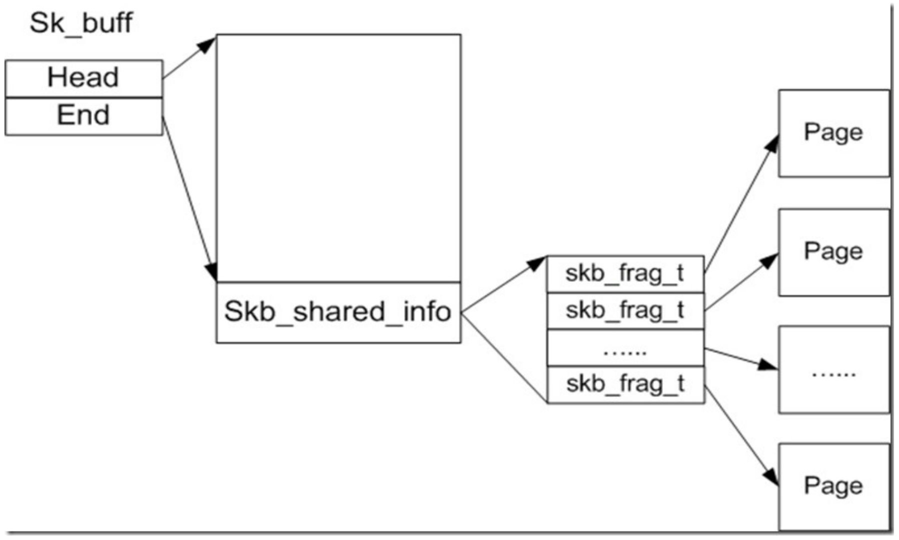
It must be understood that once paged data starts to be used on an SKB, this puts a specific restriction on all future SKB data area operations. In particular, it is no longer possible to do skb_put() operations.

The *skb_put()* modifies the tail pointer in the linear-data area to add more data.

The paged data which is mentioned above is represented by the *skb_shared_info* structure. It is at the end of the data buffer that keeps additional information about the data block.

This structure contains information about the non-linear data area for the sk_buff. By non-linear area, it means that the data contained by sk_buff are just more than that can be accommodated in the linear data area. The data contained in the non-linear data area is a continuation of the data from the offset pointed to by *end* field of the *sk_buff.* The total length of the sk_buff is stored in *len* field and the length of the non-linear (paged) data is stored in *data_len* field of *sk_buff.* The paged data area is possible only if the DMA allows scatter-gather operations on physically non-contiguous and scattered pages.
The skb_shared_info structure is used by the fragmented skbuffs. It has a meaning when the data_len field in the skbuff. header is different from zero.

This field counts the data not in the linear part of the *skbuff*. The *dataref* field counts the number of references to the fragmented part of the *skbuff*, so that a writer knows if it is necessary to copy it. The nr_frags field keeps the number of pages in which this *skbuff* is fragmented. This kind of fragmentation is done for interfaces supporting scatter and gather. This feature is described in the *netdevice* structure by the NETIF_F_SG flag. (3com 3c59x , 3com typhoon, Intel e100, ...) When an *skbuff* is to be allocated, if the *mss* is larger than a page then if the interface supports scatter and gather a linear *skbuff* of a single page is allocated with alloc_skb and then the other pages are allocated and added to the *frags* array



Representation of a complete sk_buff

1. *nr_flags* – Keeps track of number paged fragments for the *sk_buff.* It is an indication of number of elements in the *frags[]* array containing paged data for *sk_buff.*
2. *Frags[]* – this is an array of fragments containing the paged data for the sk_buff. The paged data are represented by the *skb_frag_struct.* The length of the data contained in the paged area is the sum of the number of bytes contained in each page fragemtn and is stored in the *data_len* field of the sk_buff structure.
3. *Frag_list* – This field keeps pointer to the list of *sk_buffs* representing the fragments for the original packet. If the original packet is fragmented, all the sk_buffs representing those

fragments will be linked in this list and the total length of the original sk_buff is the sum of the lengths of *skb->len* of each fragment in the *frag_list* including the length of the original sk_buff.

There are three pointers associated with the *sk_buff* that deals with managing the data.

1. Head – This field points to the start of the linear data area.
2. Data – This field points to the start of the data residing in the linear-data area. The data residing in the linear-data area may not be always start from the start of the linear-data area pointed to by head.
3. Tail – This field points to the last byte of the data residing in the linear-data area.
4. End – This field points to the end of the linear-data area and is different from *tail*. The end of the data residing in the linear-data area, so we have a tail. With this field we make sure that we dont use more than what is available.

TX Checksum offload in action

Now the device has been completely registered with the kernel. It is all set to start transmitting packets. Let us see an instance of how the checksum offloading is implemented and how the network stack communicates to the driver to instruct the device to calculate the checksum.

When a packet has to be transmitted the kernel calls the *dev_hard_start_xmit* routine. Each driver, while registering with the stack, gives a set of routines through which it can be reached from the kernel. They are stored in the *net_device* structure associated with the kernel.

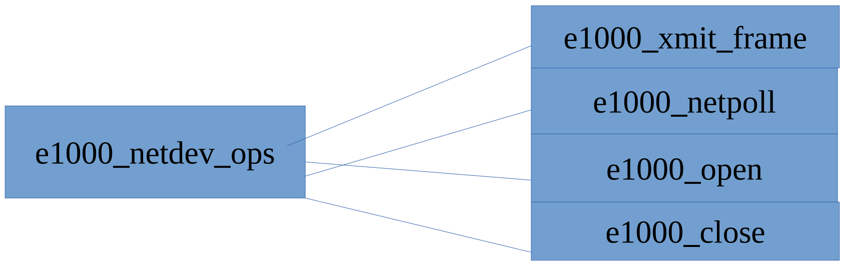


Figure showing some of the common routines of the driver

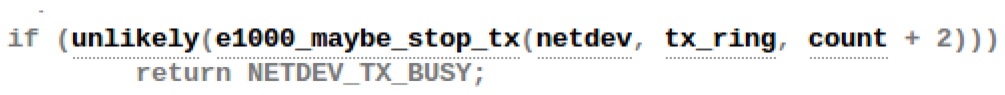
The above mentioned *e1000_netdev_ops* structure is initialized for every device. Each driver will have different routines here. This makes the task of calling the corresponding routine from the kernel, an easier job.

e1000_xmit_frame

Once *e1000_xmit_frame* is called, it is the duty of the driver to instruct the device.

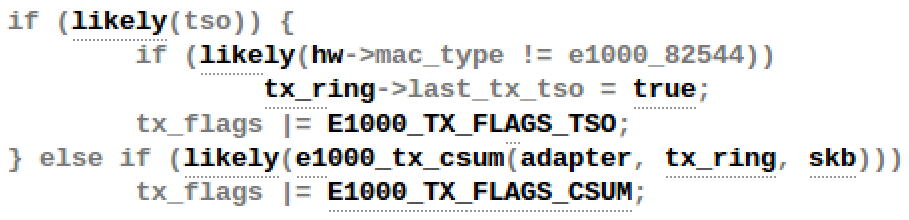
The routine gets the pointer to the ring associated with the device. It then counts the number of descriptors that are required for this transmission It calculates the number of descriptors required based on the following. The length of the data in the linear area and length of the data in the fragmented area is calculated and the total number of descriptors for this final length is calculated.

Once the number of descriptors needed is found, it also checks if those number of descriptors are available in the descriptor ring for the device to use.



```
if (unlikely(e1000_maybe_stop_tx(netdev, tx_ring, count + 2)))
        return NETDEV_TX_BUSY;
```

If everything is fine, it proceeds to find if the upper layer protocols requested any help from the device. For example, if a protocol had requested that the tcp checksum should be offloaded, decisions corresponding those requests are made.



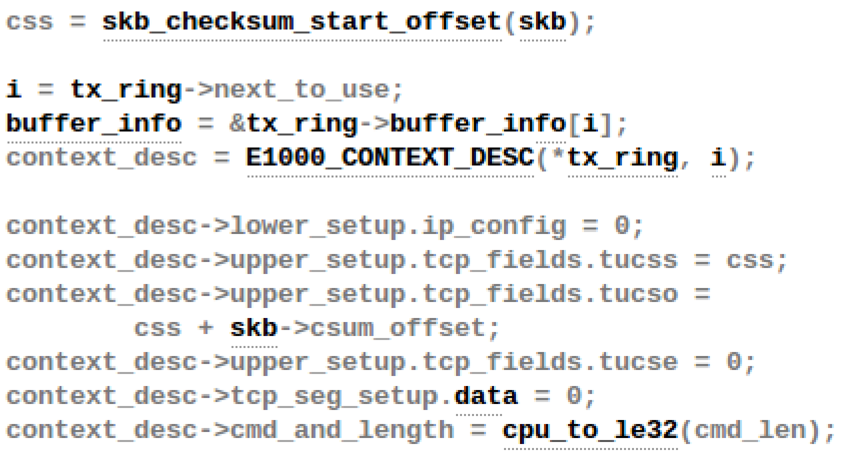
```
if (likely(tso)) {
        if (likely(hw->mac_type != e1000_82544))
                tx_ring->last_tx_tso = true;
        tx_flags |= E1000_TX_FLAGS_TSO;
} else if (likely(e1000_tx_csum(adapter, tx_ring, skb)))
        tx_flags |= E1000_TX_FLAGS_CSUM;
```

Figure illustrating the decision making process of the driver

e1000_tx_csum

The driver checks through the function *e1000_tx_csum* function if the higher level protocols have instructed the device to calculate the checksum.

As mentioned earlier, the field in the skb structure called the *summed* holds the information regarding checksum offloading. The driver will know that the stacks wants the device to calculate the checksum if *ip_summed == CHECKSUM_PARTIAL*. If the condition matches, the driver clearly tells the device of where it has to place the calculated checksum through the *checksum context descriptor.*



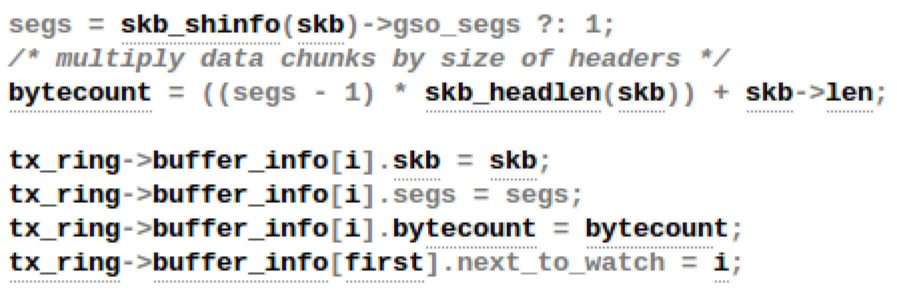
```
css = skb_checksum_start_offset(skb);

i = tx_ring->next_to_use;
buffer_info = &tx_ring->buffer_info[i];
context_desc = E1000_CONTEXT_DESC(*tx_ring, i);

context_desc->lower_setup.ip_config = 0;
context_desc->upper_setup.tcp_fields.tucss = css;
context_desc->upper_setup.tcp_fields.tucso =
        css + skb->csum_offset;
context_desc->upper_setup.tcp_fields.tucse = 0;
context_desc->tcp_seg_setup.data = 0;
context_desc->cmd_and_length = cpu_to_le32(cmd_len);
```

Figure illustrating the process involved in telling the device about the checksum

If the stack wants the device to do the checksum calculation, it sets the context descriptor for *one* time until it decides it does not want the device to perform the checksums. The NIC checks this context descriptor *every time* and then makes the decision with the data descriptor.
A new context will be loaded for example, if the stack wants to do IP checksumming. It then changes the context descriptor to indicate the hardware to do IP checksums. So the hardware *blindly* performs IP checksums for the buffers referenced by the data descriptor. It is not necessary to set a new context for each new packet. In many cases, the same checksum context can be used for a majority of the packet stream. In this case, some of the offload feature apply only for a particular traffic type, thereby avoiding all context descriptors except for the initial one. Setting this descriptor enables the device to know that it has to apply this context for all the forthcoming packets.

e1000_tx_map

This routine maps the descriptors with the buffers. It internally counts the number of descriptors needed based on the total length including the data in the linear area plus the data in the paged area. It then ties the knot at this step.



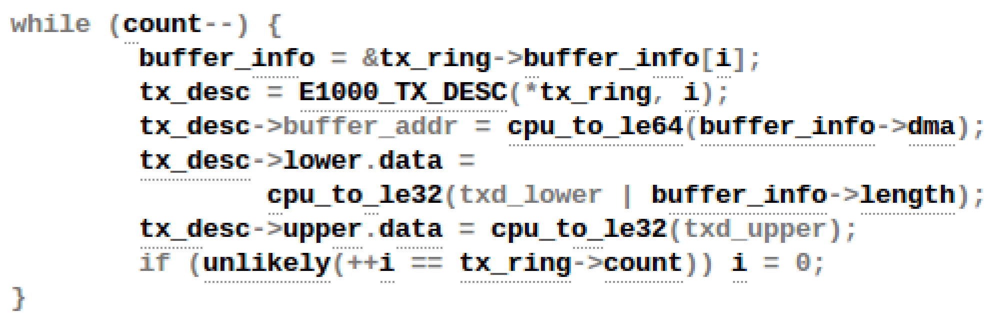
```
segs = skb_shinfo(skb)->gso_segs ?: 1;
/* multiply data chunks by size of headers */
bytecount = ((segs - 1) * skb_headlen(skb)) + skb->len;

tx_ring->buffer_info[i].skb = skb;
tx_ring->buffer_info[i].segs = segs;
tx_ring->buffer_info[i].bytecount = bytecount;
tx_ring->buffer_info[first].next_to_watch = i;
```

Figure illustrating the mapping between buffers

From the above function we see that, all the required buffers to send a packet are stored in the *buffer* structure and the starting buffer's *next_to_watch* is updated to the last buffer. So all the buffers in this range belong to a single packet.

e1000_tx_queue

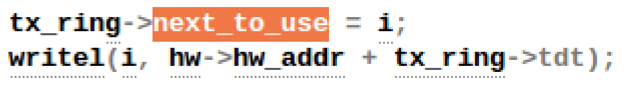
This is the actual function which sets the descriptors and informs the device about the change.



```
while (count--) {
        buffer_info = &tx_ring->buffer_info[i];
        tx_desc = E1000_TX_DESC(*tx_ring, i);
        tx_desc->buffer_addr = cpu_to_le64(buffer_info->dma);
        tx_desc->lower.data =
                cpu_to_le32(txd_lower | buffer_info->length);
        tx_desc->upper.data = cpu_to_le32(txd_upper);
        if (unlikely(++i == tx_ring->count)) i = 0;
}
```

Figure illustrating the setting of descriptors for the buffers which were mapped previosuly

The count parameter is the number of buffers needed to transmit a single packet which is supplied by the previous function. It creates a descriptor for all such buffers and stores the physical address of the buffer as indicated in the *buffer->dma* field. The following is the final step of informing the device and the point at which control leaves the software.

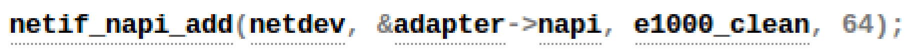


```
tx_ring->next_to_use = i;
writel(i, hw->hw_addr + tx_ring->tdt);
```

RX Checksum Offload in Action

NAPI

RX Checksumming is enabled by default in devices which support them. Whenever a packet arrives, an interrupt is raised by the device. Modern high speed networking devices follow a principle named NAPI ("New API").
If the device generates an interrupt for each and every packet, then serving this IRQ's becomes costly in terms of processor resources and time. An alternate to this mechanism can be, asking the kernel to poll the device at a frequent time to check if there are packet to process. But, if very few packets are being generated, then again it becomes overhead in terms of using CPU resources. NAPI allows dynamic switching between the modes. It goes to the poll mode when the interrupt rate it too high and to interrupt mode when the rate of input packet is low. Thus our poll function in the driver is registered.



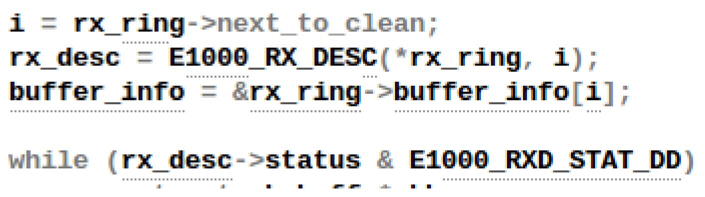
```
netif_napi_add(netdev, &adapter->napi, e1000_clean, 64);
```

Figure Illustrating the initialization of the POLL routine

e1000_cleanrx_irq

This is the main function which does the job of cleaning the RX Ring once the packet have been consumed by the kernel. It is called in the function *e1000_clean*. This function was registered as a function pointer during the configuration step.

It checks if the DD bit is set in the descriptor. The device writes the status to the descriptor and DD is a bit in the status register. If this bit it set, then it means that the packet can be taken by the software.

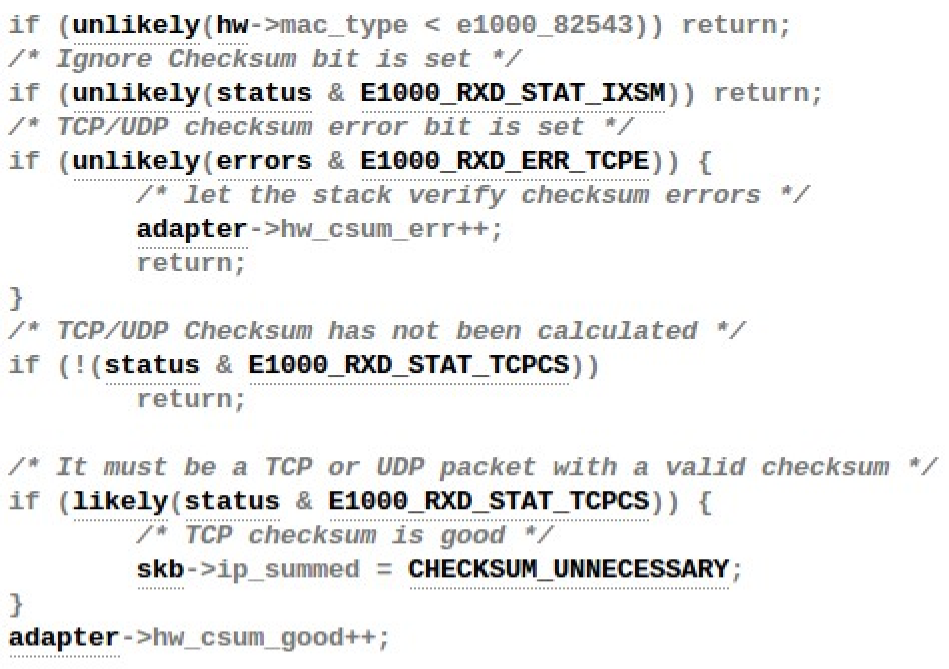


```
i = rx_ring->next_to_clean;
rx_desc = E1000_RX_DESC(*rx_ring, i);
buffer_info = &rx_ring->buffer_info[i];

while (rx_desc->status & E1000_RXD_STAT_DD)
```

Figure Illustrating the receive processing by the device driver

From the above figure we can see that the software checks if the DD bit is set in the next descriptor and if it is set, ti consumes the packet. During thr packet processing it calls *e1000_rx_checksum* which is the function responsible for examining the information given by the driver and setting the corresponding fields in the socket buffer structure.

e1000_rx_checksum



```
if (unlikely(hw->mac_type < e1000_82543)) return;
/* Ignore Checksum bit is set */
if (unlikely(status & E1000_RXD_STAT_IXSM)) return;
/* TCP/UDP checksum error bit is set */
if (unlikely(errors & E1000_RXD_ERR_TCPE)) {
        /* let the stack verify checksum errors */
        adapter->hw_csum_err++;
        return;
}
/* TCP/UDP Checksum has not been calculated */
if (!(status & E1000_RXD_STAT_TCPCS))
        return;

/* It must be a TCP or UDP packet with a valid checksum */
if (likely(status & E1000_RXD_STAT_TCPCS)) {
        /* TCP checksum is good */
        skb->ip_summed = CHECKSUM_UNNECESSARY;
}
adapter->hw_csum_good++;
```

Figure Illustrating the processing done by the driver wrto checksum offload

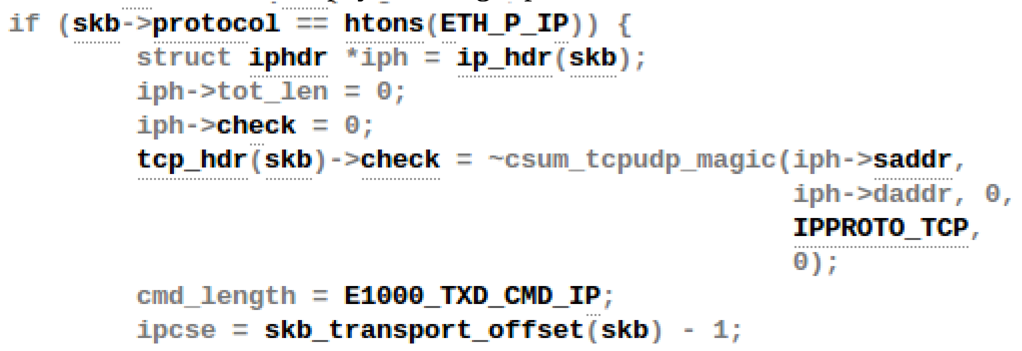
The error bits handled in the above function are explained in the receive descriptors section. If no error bit is set, the driver sets the *skb->ip_summed* field to CHECKSUM_UNNECESSARY indicating that the device has calculated the checksum and the stack need not calculate it again.

TCP Segmentation Offload in action

Hardware TCP Segmentation is one of the off-loading options of most modern TCP/IP stacks. This is often referred to as "Large Send" offloading. This feature enables the TCP/IP stack to pass to the Ethernet controller software driver a message to be transmitted that is bigger than the Maximum Transmission Unit (MTU) of the medium. It is then the responsibility of the software driver and hardware to carve the TCP message into MTU size frames that have appropriate layer 2 (Ethernet), 3 (IP), and 4 (TCP) headers. These headers must include sequence number, checksum fields, options and flag values as required. Note that some of these values (such as the checksum values) are unique for each packet of the TCP message, and other fields such as the source IP address is constant for all packets associated with the TCP message.
The offloading of these processes from the software driver to the Ethernet controller saves significant CPU cycles. The software driver shares the additional tasks to support these options with the Ethernet controller.

Transmission process with Segmentation Offload

1. The protocol stack receives from the application a block of data that is to be transmitted.
2. The stack only computes one Ethernet, IP and TCP header per segment.
3. The stack interfaces to the software driver only once per block transfer and passes the block down with the appropriate header information.
4. The driver sets up the interface to the hardware via descriptors for the TCP segmentation offload context.
5. The hardware transfers the packet data and performs the Ethernet packet segmentation and transmits based on offset and payload length parameters in the TCP/IP context descriptor.



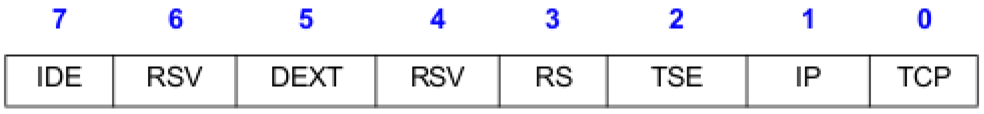
```
if (skb->protocol == htons(ETH_P_IP)) {
        struct iphdr *iph = ip_hdr(skb);
        iph->tot_len = 0;
        iph->check = 0;
        tcp_hdr(skb)->check = ~csum_tcpudp_magic(iph->saddr,
                                                 iph->daddr, 0,
                                                 IPPROTO_TCP,
                                                 0);
        cmd_length = E1000_TXD_CMD_IP;
        ipcse = skb_transport_offset(skb) - 1;
```

Figure Illustrating the calculation of pseudo headers in the socket buffer

From the above code, we can see that the pseudo header for the TCP header is calculated and stored in the checksum field of the TCP header. The driver takes the value in this field, computes the chekcsum for the TCP layer along with the TCP data and then stores the final data in the TCP header. This is how the checksum is calculated with the pseudo header.

Descriptors used for Segmentation Context

Just like setting a context descriptor for checksum offloading, we need to set a context descriptor for TCP segmentation to work. The segmentation context adds information specific to the segmentation capability. The information includes the total payload for the message, the total size of the header, the amount of the payload data and what type of protocol is used. This information is specific to the segmentation capability and is therefore ignored for context descriptors that do not have the bit corresponding to the TSO is set.



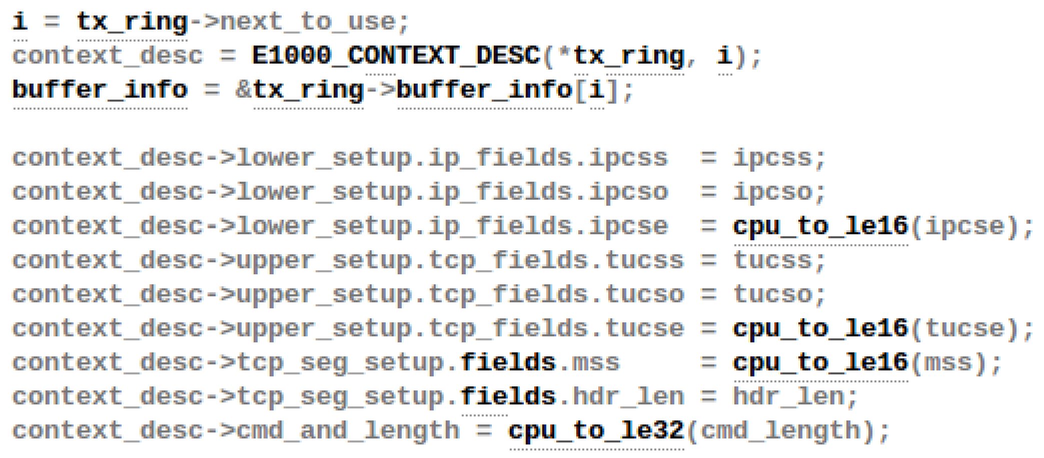
7	6	5	4	3	2	1	0
IDE	RSV	DEXT	RSV	RS	TSE	IP	TCP

Figure Showing the Command field in the context descriptor

Setting the TSE bit in the command field indicates that the descriptor refers to the TCP segmentation context. This causes the checksum offloading, packet length, header length and maximum segment size parameters to be loaded from the descriptor to the Ethernet controller.
If we revisit the structure of the context descriptor mentioned in the Descriptors section, we can see that the TCP/IP context descriptor contains the following fields which are used for TCP segmentation Offload.
MSS – Tells the maximum segment size. I.e it indicates the maximum TCP or UDP payload segment sent per frame, not including any header. The total length of each frame sent by the TCP segmentation mechanism is MSS + HRDLEN bytes.
HDRLEN – Specifies the length of the header to be used for each frame of a TCP segmentation operation. The first HDRLEN bytes fetched from the data descriptor are stored internally and used as a prototype header for each section and prepended to each payload segment to form individual frames. For UDP packets this is normally equal to “UDP checksum offset + 2 bytes”. For TCP packets it is normally equal to “TCP checksum offset + 4 + TCP header option bytes”.
PAYLEN – The packet length field is the total number of payload bytes for this TCP segmentation offload context. I.e the total number of payload bytes that could be distributed across multiple frames after TCP segmentation is performed.
Once the TCP segmentation context has been set, the next descriptor provides the initial data to transfer.

e1000_tso

For a device to support TSO, it also needs to support Scatter-gather and Checksum offloading. This is the main function which sets the context descriptor for TSO as discussed above. When the TSO feature is advertised by the device, the driver will receive super-sized frames. This is indicated to the driver by *skb_info(skb)->gso_size* being non-zero. The variable *gso_size* indicates the size with which the hardware should fragment the TCP data.



```
i = tx_ring->next_to_use;
context_desc = E1000_CONTEXT_DESC(*tx_ring, i);
buffer_info = &tx_ring->buffer_info[i];

context_desc->lower_setup.ip_fields.ipcss  = ipcss;
context_desc->lower_setup.ip_fields.ipcso  = ipcso;
context_desc->lower_setup.ip_fields.ipcse  = cpu_to_le16(ipcse);
context_desc->upper_setup.tcp_fields.tucss = tucss;
context_desc->upper_setup.tcp_fields.tucso = tucso;
context_desc->upper_setup.tcp_fields.tucse = cpu_to_le16(tucse);
context_desc->tcp_seg_setup.fields.mss     = cpu_to_le16(mss);
context_desc->tcp_seg_setup.fields.hdr_len = hdr_len;
context_desc->cmd_and_length = cpu_to_le32(cmd_length);
```

Figure Illustrating the fields of context descriptor being set

The above piece of code, shows how the fields of the context descriptor are set to enable the segmentation context. Once this context descriptor is set for the TCP Segmentation, then the driver passes on the super blocks to the NIC which does the actual segmentation. The rest of the packet flow is same as discussed for the checksum offload context.

Large Receive Offload in Action (GRO)

This is analogous to the functioning of TSO. Just like how Linux uses TSO to pass a large buffer to the driver and let the device do the checksumming and other calculations, LRO allows for the OS to receive aggregated packets from the device. I.e LRO combines received TCP packets to a single larger TCP packet and passes them to the network stack in order to increase the performance.

The problem with large receive offload is that, it merges everything. This can be faulty at some times. If there are important information in the headers which comes in between the packet, for example, RST flag in TCP headers, then such vital information will be lost. The solution to this was using Generic Receive Offload. Here, the criteria for which the packets can be merged is greatly restricted. For example, the MAC address should be identical, only a few TCP / IP headers can differ, the TCP timestamps must be equal etc. With these constraints, the merged packets can be re segmented losslessly.

The main idea behind GRO is that, we can accumulate consecutive packets into one huge packet and then process the whole group as one packet.

Each NAPI instance maintains a list of GRO packets which are to be accumulated. It stores the address in *napi->gro_list.* This is then dispatched to the network layer protocol that the packet is destined for. Each network layer that supports GRO implements, *gro_receive* and *gro_complete.*

gro_receive attempts to match the incoming skb with the ones that have already been queued onto the *gro->list.* The *gro_complete* method is invoked once we have committed to receiving a GRO skb. It is in this method, a collection of individual packets look truly like a huge packet. Checksums are updated, as are various flags in the head 'skb' given to the network stack.

e1000_receive_skb

After packet processing from *e1000_clean_rx_irq*, the actual extraction of the contents is done in *e1000_receive_skb*. It calls the gro receive function and passes the NAPI instance associated with this device and skb of the received packet.

dev_gro_receive

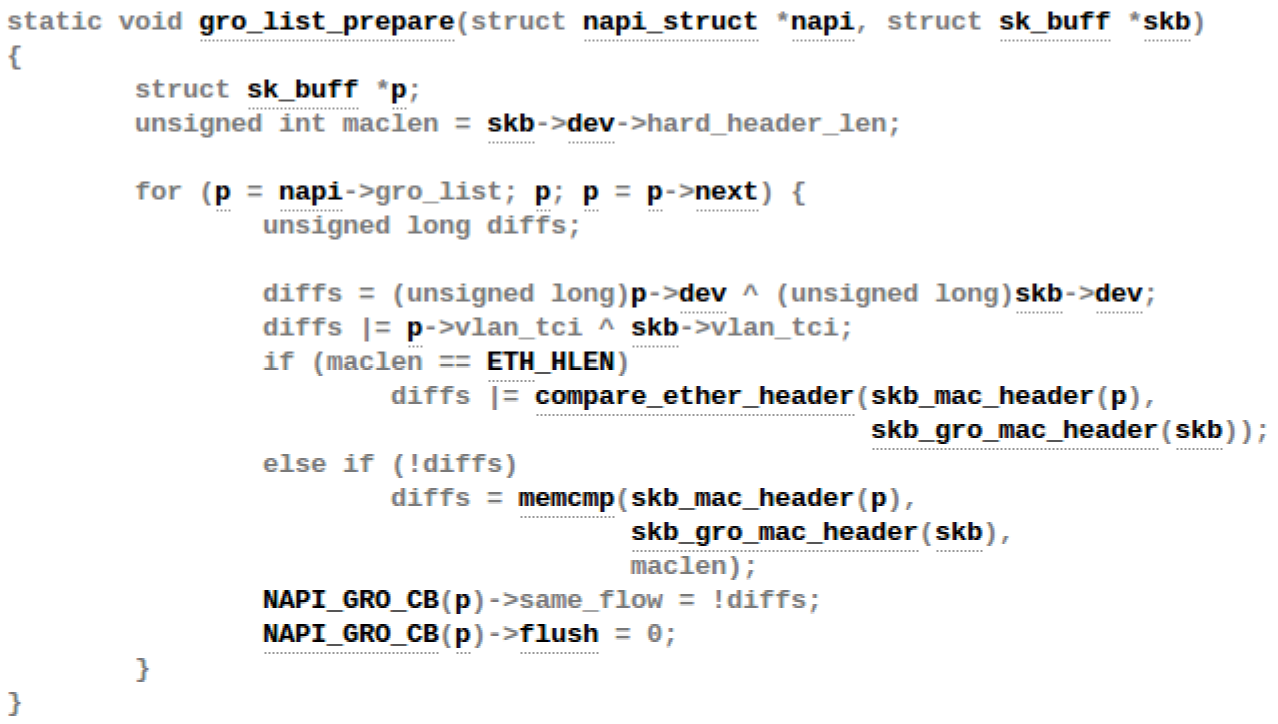
This function takes care of preparing the gro segments into one big packet and sending it up the stack. It calls the associated protocol's gro receive function.

It first checks if the feature flag *NETIF_F_GRO* is set. If the feature flag is not set, then it means that the device does not support GRO and hence the control goes to the normal receive path.

It then checks if *netpoll* is on or not. With NAPI, packets are accumulated into a big one, only on the next poll. I.e The packets are not tried to be merged on every interrupt. Instead, whenever the NAPI's poll takes place the packets are merged into on. This check is being done by *netpoll_rx_on.*

It also checks if the generic segmentation offload is enabled and looks if the skb has paged data. If either one of these is true, then again normal processing is done.

It parses through the skb's and compares this skb with them to check if they are of the same flow or not.



```
static void gro_list_prepare(struct napi_struct *napi, struct sk_buff *skb)
{
        struct sk_buff *p;
        unsigned int maclen = skb->dev->hard_header_len;

        for (p = napi->gro_list; p; p = p->next) {
                unsigned long diffs;

                diffs = (unsigned long)p->dev ^ (unsigned long)skb->dev;
                diffs |= p->vlan_tci ^ skb->vlan_tci;
                if (maclen == ETH_HLEN)
                        diffs |= compare_ether_header(skb_mac_header(p),
                                                      skb_gro_mac_header(skb));
                else if (!diffs)
                        diffs = memcmp(skb_mac_header(p),
                                       skb_gro_mac_header(skb),
                                       maclen);
                NAPI_GRO_CB(p)->same_flow = !diffs;
                NAPI_GRO_CB(p)->flush = 0;
        }
}
```

Figure Illustrating the function which scans the SKB list and checks for same flow

The *same_flow* value is 1, for packets belonging to the same flow. *dev_gro_receive* calls this function to determine the packets which belong to the same flow.

Conclusion

This prototype is exactly similar and is also followed by the NF NIC. Thus, an in-depth study of this also illuminates the happenings inside the NF NIC. This also makes design process simpler, as we now know how an industry standard device works inside the Linux kernel. The above study will be used as the base for making design changes in NetFPGA's NF NIC and similar performance will also be tried to be achieved.

References

1. “*Understanding Linux Network Internals*” by O'Reilly publications.

2. “*TCP/IP Architecture, Design and Implementation in Linux*” by IEEE Computer Society

3. “*How SKB's work*” http://vger.kernel.org/~davem/skb_data.html

4. “*Management of sk_buffs*” http://people.cs.clemson.edu/~westall/853/notes/skbuff.pdf

5. “*Linux cross reference”* http://lxr.free-electrons.com