1.[<linux/types.h](http://stackoverflow.com/questions/20302860/supporting-linux-types-h-osx)> :

It’s kernel Header file used to define some inbuilt kernel types like '\_\_s32,

type name '\_\_s32'

type name '\_\_u8'

type name '\_\_u16’

**2. <linux/skbuff.h> :**

Definitions for the 'struct sk\_buff' memory handlers.

struct sk\_buff \* next; /\* Next buffer in list \*/

struct sk\_buff \* prev; /\* Previous buffer in list \*/

struct sk\_buff\_head \* list; /\* List we are on \*/

***Checksumming of received packets by device.***

***The skb was already checksummed by the protocol, or a checksum is not required.***

**3.<linux/icmp.h> :**

icmp - Linux IPv4 ICMP kernel module.

This kernel protocol module implements the Internet Control Message

Protocol defined in RFC 792. It is used to signal error conditions

and for diagnosis. The user doesn't interact directly with this

module; instead it communicates with the other protocols in the

kernel and these pass the ICMP errors to the application layers. The

kernel ICMP module also answers ICMP requests.

A user protocol may receive ICMP packets for all local sockets by

opening a raw socket with the protocol **IPPROTO\_ICMP**. See [raw(7)](http://man7.org/linux/man-pages/man7/raw.7.html) for

more information. The types of ICMP packets passed to the socket can

be filtered using the **ICMP\_FILTER** socket option. ICMP packets are

always processed by the kernel too, even when passed to a user

socket.

Linux limits the rate of ICMP error packets to each destination.

**ICMP\_REDIRECT** and **ICMP\_DEST\_UNREACH** are also limited by the

destination route of the incoming packets.

**Eg:**

*icmp\_destunreach\_rate*

**4. <linux/slab.h> :**

***This delays freeing the SLAB page by a grace period, it does \_NOT\_delay object freeing. This means that if you do kmem\_cache\_free()that memory location is free to be reused at any time. Thus it may* *be possible to see another object there in the same RCU grace period.This feature only ensures the memory location backing the object* *stays valid, the trick to using this is relying on an independent* *object validation pass.***

***This is useful if we need to approach a kernel structure obliquely,from its address obtained without the usual locking. We can lock* *the structure to stabilize it and check it's still at the given address,only if we can be sure that the memory has not been meanwhile reused* *for some other kind of object (which our subsystem's lock might corrupt).***

***ZERO\_SIZE\_PTR will be returned for zero sized kmalloc requests.***

**#ifdef CONFIG\_SLAB**

***The largest kmalloc size supported by the SLAB allocators is***

***32 megabyte (2^25) or the maximum allocatable page order if that is* *\* less than 32 MB.***

These routines are used to dynamically request pointer-aligned chunks of memory, like malloc and free do in userspace, but kmalloc() takes an extra flag word. Important values:

GFP\_KERNEL

May sleep and swap to free memory. Only allowed in user context, but is the most reliable way to allocate memory.

GFP\_ATOMIC

Don't sleep. Less reliable than GFP\_KERNEL, but may be called from interrupt context. You should *really* have a good out-of-memory error-handling strategy.

GFP\_DMA

Allocate ISA DMA lower than 16MB. If you don't know what that is you don't need it. Very unreliable.

If you see a sleeping function called from invalid context warning message, then maybe you called a sleeping allocation function from interrupt context without GFP\_ATOMIC. You should really fix that. Run, don't walk.

If you are allocating at least PAGE\_SIZE (include/asm/page.h) bytes, consider using \_\_get\_free\_pages() (include/linux/mm.h). It takes an order argument (0 for page sized, 1 for double page, 2 for four pages etc.) and the same memory priority flag word as above.

If you are allocating more than a page worth of bytes you can use vmalloc(). It'll allocate virtual memory in the kernel map. This block is not contiguous in physical memory, but the MMU makes it look like it is for you (so it'll only look contiguous to the CPUs, not to external device drivers). If you really need large physically contiguous memory for some weird device, you have a problem: it is poorly supported in Linux because after some time memory fragmentation in a running kernel makes it hard. The best way is to allocate the block early in the boot process via the alloc\_bootmem() routine.

Before inventing your own cache of often-used objects consider using a slab cache in include/linux/slab.h

**5.<net/ip.h>:**

**Refer pdf on it.**

**6.<net/udp.h>:**

**It’s give protocal implementation functions.**