

Networking History

(Reference: Alan Cox¹ Homepage)

Ross Biro wrote the original kernel based networking code for Linux.

He used ethernet drivers written by Donald Becker, a SLIP driver written by Laurence Culhane and a D-Link(?) driver by Bjorn Ekwall.

The further development of the Linux networking code was later taken up by Fred van Kempen, who took Ross's code and produced the NET-2 release of network code. NET-2 went through a number of revisions until release NET-2d, when Alan Cox set about debugging Fred's code with the aim of producing a stable and working release of code for incorporation into the standard kernel releases.

This code was called originally called NET-2D(ebugged) and was incorporated into the standard kernel releases some time before Linux version 1.0 was released.

NET-3 Was released with the 1.2.x Linux version. And The latest version of the code, **NET-4** appears in kernel releases 2.x and later.

¹ <http://www.linux.org.uk>

Layers of the implementation

The IP protocol stack is generally considered to consist of three *kernel* layers:

Transport layer

Network layer

Link layer

The linux network implementation is designed to support multiple protocol stacks operating concurrently all within the familiar *socket API*. To do this efficiently the *transport* and *link* layers are broken into identifiable sub-components, and as one traverses the protocol stacks from top to bottom the implementation becomes "wider" then narrower and then "wider" again.

In the following discussion "protocol stack", "protocol family", and "network architecture" mean the same thing,

Elements of the protocol stacks

<i>socket</i>	A thin layer with a single implementation instance that serves all protocol families.
<i>protocol_family</i>	A functional layer with one instance per protocol family (e.g. PF_INET, PF_ATMPVC, PF_X25, etc.) It contains the generic components of the transport layer each protocol family. Thus, the architecture expects one instance per family.
<i>transport_protocol</i>	For each protocol family there is one instance of the this layer for each transport protocol. It contains the esoteric components of the particular transport (e.g., UDP, TCP, etc.). Your project will be to build one of these.
<i>network_layer</i>	One instance of this layer is expected for each protocol family. For PF_INET this component contains the implementation of IPV4. The terminology and architecture get somewhat blurred by IPV4/IPV6 and ATMPVC/ATMSVC components.
<i>generic_device(dev)</i>	There is a single instance of the generic device layer that service all protocol families. It contains the NIC independent aspects of the MAC layer protocol, output packet scheduling, and demultiplexing of input packets with delivery to the proper network layer.
<i>device class</i>	There is one instance of this layer for each device class (e.g., Ethernet, ATM, token ring, etc.) It contains components that are common to the MAC protocol of the device class but independent of the requirements of any specific NIC.
<i>device driver</i>	There is one instance of this layer for each specific NIC or family of NICs (e.g. 3C59x, e100, e1000, sk98lin)

Networking Initialization in Linux

The *init* process is a *kernel thread* created during the latter stages of the system boot procedure. It runs as the function *init()* which is defined in *init/main.c* and drives the initialization of those kernel components that require a process context. Network initialization begins in the call to the *do_basic_setup()* function.

```
836 static int __init kernel_init(void * unused)
837 {
838     lock_kernel();
839     /*
840      * init can run on any cpu.
841      */
842     set_cpus_allowed_ptr(current, CPU_MASK_ALL_PTR);
843     /*
844      * Tell the world that we're going to be the grim
845      * reaper of innocent orphaned children.
846      *
847      * We don't want people to have to make incorrect
848      * assumptions about where in the task array this
849      * can be found.
850      */
851     init_pid_ns.child_reaper = current;
852
853     cad_pid = task_pid(current);
854
855     smp_prepare_cpus(setup_max_cpus);
856
857     do_pre_smp_initcalls();
858     start_boot_trace();
859
860     smp_init();
861     sched_init_smp();
862
863     cpuset_init_smp();
864
865     do_basic_setup();
```

The *do_basic_setup()* function

The *do_basic_setup()* function which is also defined in *init/main.c* performs some ad hoc early initialization and then calls *do_initcalls()* to invoke subsystem initializers that have registered themselves.

```
762 * Ok, the machine is now initialized. None of the devices
763 * have been touched yet, but the CPU subsystem is up and
764 * running, and memory and process management works.
765 *
766 * Now we can finally start doing some real work..
767 */
768 static void __init do_basic_setup(void)
769 {
770     rcu_init_sched(); /* needed by module_init stage. */
771     init_workqueues();
772     usermodehelper_init();
773     driver_init();
774     init_irq_proc();
```

Finally, the *do_initcalls()* function actually drives much of the protocol and general system initialization in a clever, but not obvious way.

```
775     do_initcalls();
776 }
```

The *do_initcalls()* function

This function is table driven. For a component to enter its entry point into the call table it includes the following macro which adds the specified entry point (here *sock_init*) to the *initcall* table. Formerly, in 2.4 kernels there was only a single *initcall()* macro. Now there are several.

Hence, as the *initcall* table is processed, the pointer *call* on lines 754 and 755 eventually points to the *sock_init()* function and that is how *sock_init()* and the other component initializers are called.

Line 2240 below is in the module *socket.c* which contains the function *sock_init()*.

```
2240 core_initcall(sock_init);          /* early initcall */

748 extern initcall_t __initcall_start[], __initcall_end[],
      __early_initcall_end[];
749
750 static void __init do_initcalls(void)
751 {
752     initcall_t *call;
753
754     for (call = __early_initcall_end; call < __initcall_end; call++)
755         do_one_initcall(*call);
756
757     /* Make sure there is no pending stuff from the initcall sequence */
758     flush_scheduled_work();
759 }

81 /* initcalls are now grouped by functionality into separate
82  * subsections. Ordering inside the subsections is determined
83  * by link order.
84  * For backwards compatibility, initcall() puts the call in
85  * the device init subsection.
86  */
87
88 #define __define_initcall(level,fn) \
89     static initcall_t __initcall_##fn __attribute_used__ \
90     __attribute__((__section__(".initcall" level ".init"))) = fn
91
92 #define core_initcall(fn)    __define_initcall("1",fn)
```

Subsystem, device driver, and protocol initialization

The initialization of the protocol (and other) modules is driven by the *do_initcalls()* function. The following code taken from the 2.4 kernel family illustrates the general idea. The general idea remains the same in 2.6 but the grubby details are even grubbier.

Here *__initcall_start* points to a table of function pointers.

Each function pointer points to the initialization entry point of some kernel component that needs to initialize itself.

The initialization calls are executed sequentially until *__initcall_end* is reached.

```
648 static void __init do_initcalls(void)
649 {
650     initcall_t *call;
651
652     call = &__initcall_start;
653     do {
654         (*call)();
655         call++;
656     } while (call < &__initcall_end);
657
658     /* Make sure there is no pending stuff from the
659        initcall sequence */
659     flush_scheduled_tasks();
660 }
```

Details of the initialization mechanism

The mechanism by which the initialization pointers enter the table is somewhat subtle, and will be illustrated via *inet_init()*.

Its objectives are to:

- Automatically add a pointer to the function to the call table used by *do_initcalls()*
- Place both the function itself and the pointer into sections located at the end of the kernel.
- Allow the function to be compiled as either part of the kernel or as part of a module with no source changes.

The *inet_init()* function defined in `net/ipv4/af_inet.c` is the AF_INET Protocol initialization function.... *Note that the comment is now erroneous.*

```
1105 /*
1106  *      Called by socket.c on kernel startup.
1107  */
1108
1109 static int __init inet_init(void)
1110 {
1111     printk(KERN_INFO"NET4:Linux TCP/IP 1.0 for NET 4.0\n");
1112     :
1201 }
1202 module_init(inet_init);
```

Put the body of the function in the *text.init* section

Place a pointer to *inet_init()* into the call table.

The file `include/linux/init.h` provides insight into the working of `do_initcalls`².

Two elements are of particular interest here

- the `__init` attribute:
 - causes the function itself to be assigned to a code section named `.text.init`;
 - when the kernel is linked `.text.init` is placed at the end of the kernel code;
 - the space it occupies is then freed at the completion of initialization.

```
static int __init inet_init(void) yields

static int __attribute__((__section__ (".text.init")))
    inet_init(void)
{
    ...
    printk(KERN_INFO "NET4: Linux TCP/IP 1.0 for NET4.0\n");
    ...
}
```

- the `module_init(inet_init)` macro call:
 - generates a variable named `__initcall_inet_init` (in this case)
 - assigns the variable to the section `.initcall.init`
 - assigns the address of `inet_init` as the value of `__initcall_inet_init`

```
module_init(inet_init)yields
initcall_t __initcall_inet_init
    __attribute__((unused,__section__ (".initcall.init"))) =
    inet_init;
```

The `__init` attribute and the `module_init()` macro work much differently if the symbol `MODULE` is defined.

This difference is the key to permitting a component to be compiled into the kernel or compiled separately *with no source changes*.

² http://www.skyfree.org/linux/kernel_network/startup.html
(Provides A Good Explanation on How Kernel Network Initialization Takes Place.)

Details of the expansion

The macros themselves are defined in *include/linux/init.h*. The `##fn` in the definition of the `__initcall` macro is used to indicate the the characters *fn* do indeed represent the macro argument and should be replaced by the value used in the call to the macro.

```
41 #ifndef MODULE
42
43 #ifndef __ASSEMBLY__
44
45 /*
46 * Used for initialization calls..
47 */
48 typedef int (*initcall_t)(void);
49 typedef void (*exitcall_t)(void);
50
51 extern initcall_t __initcall_start, __initcall_end;
52
53 #define __initcall(fn) \
54 static initcall_t __initcall_##fn __init_call = fn
55
56
57
58 #define __init __attribute__((__section__(
59     ".text.init")))
60
61
62
63 #define __initdata __attribute__((__section__(
64     ".data.init")))
65
66
67
68 #define __init_call __attribute__((unused,__section__(
69     ".initcall.init")))
70
71
72
73
74
75
76
77
78
79
80 #define __initdata __attribute__((__section__(
81     ".data.init")))
82
83 #define __init_call __attribute__((unused,__section__(
84     ".initcall.init")))
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101 #define module_init(x) __initcall(x);
```

Notes:

In line 101 we see that `module_init(inet_init)` expands to
`__initcall(inet_init)`.

In line 53 we see that `__initcall(inet_init)` itself expands to
`initcall_t __initcall_inet_init __init_call = inet_init`.

which further expands via line 83 to

`initcall_t __initcall_inet_init __attribute__((unused,__section__(
".initcall.init"))) = inet_init;`

Section management

The material covered so far explained:

How entries are assigned to the table of *initcall* function pointers

It did not explain:

How the table came to be placed at the end of the kernel

The location of the `__initcall_start` and `__initcall_end` values delimit the table

These functions are accomplished via the linking script *arch/i386/vmlinux.lds*

```
36 _edata = .;                /* End of data section */
37
38 . = ALIGN(8192);           /* init_task */
39 .data.init_task : { *(.data.init_task) }
40
41 . = ALIGN(4096);          /* Init code and data */
42 __init_begin = .;
43 .text.init : { *(.text.init) }
44 .data.init : { *(.data.init) }
45 . = ALIGN(16);
46 __setup_start = .;
47 .setup.init : { *(.setup.init) }
48 __setup_end = .;
49 __initcall_start = .;
50 .initcall.init : { *(.initcall.init) }
51 __initcall_end = .;
52 . = ALIGN(4096);
53 __init_end = .;
54
```

The *.initcall.init* table of function pointers

Linking the kernel with the map option shows how it all works out:

Note that the table of function pointers is densely packed with each pointer consuming 4 bytes of storage:

```
linux.map
          0xc029f4e8
.initcall.init 0xc029f4e8    0x90    __initcall_start=.
*(.initcall.init)
.initcall.init
          0xc029f4e8    0xc    arch/i386/kernel/kernel.o
          0xc029f4e8    __initcall_dmi_scan_machine
          0xc029f4ec    __initcall_cpuid_init
          0xc029f4f0    __initcall_apm_init
.initcall.init
          0xc029f4f4    0x4    kernel/kernel.o
          0xc029f4f4    __initcall_uid_cache_init
.initcall.init
          0xc029f4f8    0xc    mm/mm.o
          0xc029f4f8    __initcall_kmem_cpucache_init
          0xc029f4fc    __initcall_kswapd_init
          0xc029f500    __initcall_init_shmem_fs
.initcall.init
          0xc029f504    0x3c   fs/fs.o
          0xc029f504    __initcall_bdflush_init
          0xc029f508    __initcall_init_pipe_fs
          0xc029f50c    __initcall_fasync_init
          0xc029f520    __initcall_init_elf_binfmt
          0xc029f524    __initcall_init_proc_fs
          0xc029f528    __initcall_partition_setup
          0xc029f52c    __initcall_init_ext2_fs
          0xc029f530    __initcall_init_fat_fs
          0xc029f534    __initcall_init_msdos_fs
          0xc029f538    __initcall_init_iso9660_fs
          0xc029f53c    __initcall_init_reiserfs_fs
```

```

.initcall.init      0xc029f540      0x4 drivers/block/block.o
                   0xc029f540      __initcall_loop_init
.initcall.init      0xc029f544      0x4 drivers/char/char.o
                   0xc029f544      __initcall_rs_init
.initcall.init      0xc029f548      0x8 drivers/net/net.o
                   0xc029f548      __initcall_dummy_init_module
                   0xc029f54c      __initcall_rtl8139_init_module
.initcall.init      0xc029f550      0x4 drivers/ide/idedriver.o
                   0xc029f550      __initcall_ide_cdrom_init
.initcall.init      0xc029f554      0x4 drivers/cdrom/driver.o
                   0xc029f554      __initcall_cdrom_init
.initcall.init      0xc029f558      0x4 drivers/pci/driver.o
                   0xc029f558      __initcall_pci_proc_init
.initcall.init      0xc029f55c      0x1c net/network.o
                   0xc029f55c      __initcall_p8022_init
                   0xc029f560      __initcall_snap_init
                   0xc029f564      __initcall_inet_init
                   0xc029f568      __initcall_af_unix_init
                   0xc029f56c      __initcall_netlink_proto_init
                   0xc029f570      __initcall_packet_init
                   0xc029f574      __initcall_atalk_init
                   0xc029f578      __initcall_end=.

```

Initialization of installable modules

Many protocols (but not inet) may also be built as installable modules. If the symbol *MODULE* is defined, then the following definitions hold:

```
113
114 #else    /* MODULE */
115
116 #define __init
117 #define __exit
118 #define __initdata
119 #define __exitdata
120 #define __initcall(fn)

131 typedef int (*__init_module_func_t)(void);
132 typedef void (*__cleanup_module_func_t)(void);
133 #define module_init(x) \
134 int init_module(void) __attribute__((alias(#x))); \
135 static inline __init_module_func_t __init_module_inline(void) \
136 { return x; }
```

Notes:

In this case the *__init*, *__initdata*, and *__initcall* attributes are null.

The *module_init(x)* macro expands into an *init_module()* function which invokes *x* in an inline expansion.

For example, when *MODULE* is defined,

```
module_init(fw_init);
```

expands as shown in lines 133 - 136 above to yield

```
int init_module(void) __attribute__((alias("fw_init")));
static inline __init_module_func_t
__init_module_inline(void) { return fw_init; };
```

Supported address families (or socket types)

The supported address families are defined in *linux/include/linux/socket.h*. There are now 36 of these in kernel 2.6.28!

```
158/* Supported address families. */
159#define AF_UNSPEC      0
160#define AF_UNIX        1      /* Unix domain sockets      */
161#define AF_LOCAL       1      /* POSIX name for AF_UNIX   */
162#define AF_INET        2      /* Internet IP Protocol     */
163#define AF_AX25        3      /* Amateur Radio AX.25      */
164#define AF_IPX         4      /* Novell IPX                */
165#define AF_APPLETALK   5      /* AppleTalk DDP            */
166#define AF_NETROM      6      /* Amateur Radio NET/ROM    */
167#define AF_BRIDGE      7      /* Multiprotocol bridge     */
168#define AF_ATMPVC      8      /* ATM PVCs                  */
169#define AF_X25         9      /* Reserved for X.25 project */
170#define AF_INET6       10     /* IP version 6             */
171#define AF_ROSE        11     /* Amateur Radio X.25 PLP    */
172#define AF_DECnet      12     /* Reserved for DECnet project */
173#define AF_NETBEUI     13     /* Reserved for 802.2LLC project */
174#define AF_SECURITY    14     /* Security callback pseudo AF */
175#define AF_KEY         15     /* PF_KEY key management API */
176#define AF_NETLINK     16
177#define AF_ROUTE       AF_NETLINK /* Alias to emulate 4.4BSD */
178#define AF_PACKET      17     /* Packet family            */
179#define AF_ASH         18     /* Ash                      */
180#define AF_ECONET      19     /* Acorn Econet             */
181#define AF_ATMSVC      20     /* ATM SVCs                 */
182#define AF_SNA         22     /* Linux SNA Project (nutters!) */
183#define AF_IRDA        23     /* IRDA sockets             */
184#define AF_PPPOX       24     /* PPPoX sockets           */
185#define AF_WANPIPE     25     /* Wanpipe API Sockets */
186#define AF_LLC         26     /* Linux LLC                */
187#define AF_CAN         29     /* Controller Area Network  */
188#define AF_TIPC        30     /* TIPC sockets             */
189#define AF_BLUETOOTH   31     /* Bluetooth sockets       */
190#define AF_MAX         32
```

There are now 36 protocols in kernel 2.6.28!

PF_INET versus AF_INET

Technically, the term PF_INET refers to the protocol family and AF_INET refers to the address family. Historically these were distinguished so as to enable one protocol family to support several address types. This hasn't happened. Now the terms AF (address family) and PF (protocol family) are synonymous in Linux and the PF_ values are directly derived from their AF_ counterparts in *include/linux/socket.h* as:

```
167 #define PF_INET AF_INET  
    etc.
```

The *sock_init()* function

The *sock_init()* function, defined in `net/socket.c`, once played a prominent role in initialization but that role is now considerably diminished.

The comment regarding the sock SLAB cache is now defective.

The present day *sk_init()* only sets up default send and receive buffer quotas. *Each transport protocol is now responsible for creating its own SLAB cache of stuck sock.*

The call to *skb_init()* creates the *standard and fclone buffer header caches.*

```
2209 static int __init sock_init(void)
2210 {
2211     /*
2212      *      Initialize sock SLAB cache.
2213      */
2214
2215     sk_init();
2216
2217     /*
2218      *      Initialize skbuff SLAB cache
2219      */
2220     skb_init();
2221
2222     /*
2223      *      Initialize the protocols module.
2224      */
2225
```

The *sock_init()* function is also responsible for creating a cache of *socket* and *inode* structures and registering the socket virtual file system. The call to *init_inode_cache* creates a [cache](#) of structures from which sockets and inodes will be allocated.

```
2226     init_inodecache();
2227     register_filesystem(&sock_fs_type);
2228     sock_mnt = kern_mount(&sock_fs_type);
2229
2230     /* The real protocol initialization is performed in later
2231        initcalls.
2232     */
2233     #ifdef CONFIG_NETFILTER
2234         netfilter_init();
2235     #endif
2236
2237     return 0;
2238 }
```

The `sk_init()` function

The networking layer uses the slab allocator to create caches of all fixed size data structures that are dynamically allocated and freed. Once upon a time the `sk_init()` function was called to create a cache of *struct sock* types. Now this is done on a per-transport protocol basis. We will see that both *struct socket* and *struct sock* are important elements of the protocol implementation. The former is generic and supports all address families. The latter was designed to be specific to the IP stack but may also be used by other families as well.

Now all `sk_init()` does is initialize the buffer write (*wmem*) and read (*rmem*) buffer quotas associated with each socket. Variables prefixed with `sysctl` are mapped in the `/proc` file system and may be modified by the system administrator during system operation.

The values being defined here reside in `/proc/sys/net/core`

```
1099 void __init sk_init(void)
1100 {
1101     if (num_physpages <= 4096) {
1102         sysctl_wmem_max = 32767;
1103         sysctl_rmem_max = 32767;
1104         sysctl_wmem_default = 32767;
1105         sysctl_rmem_default = 32767;
1106     } else if (num_physpages >= 131072) {
1107         sysctl_wmem_max = 131071;
1108         sysctl_rmem_max = 131071;
1109     }
1110 }
1111
```

Cache allocations and the efficient management of kernel memory objects

The kernel often needs to keep large arrays of data structures having fixed size but...

The fixed size of the structure is not often page size.

Having to interact with the buddy system for each entity allocated

- wastes space (if the objects are much smaller than page size)
- wastes time since the buddy system is not especially efficient.

The slab allocator, introduced in Solaris, provides a solution

The *slab allocator* pre-allocates *caches*

Each *cache* contains one or more *slabs*

Each *slab* is a multiple of page size.

Each *slab* holds multiple objects

Slabs are maintained in three lists:

- | | |
|-----------|------------------------------------|
| Empty - | All objects allocated |
| Partial - | Some but not all objects allocated |
| Full - | No objects allocated |

The buddy system gets involved only when slabs are allocated or freed

The state of the slab allocator

/proc/slabinfo provides the state of the slab allocator.

Information elements include:

cache-name, num-active-objs, total-objs, object size
num-active-slabs, total-slabs, num-pages-per-slab

Name	Active obj	#obj	Sz Obj	ActSlb	#Slb	#Pg		
kmem_cache	68	68	232	4	4	1	:	252 126
clip_arp_cache	0	0	128	0	0	1	:	252 126
ip_conntrack	172	253	352	22	23	1	:	124 62
tcp_tw_bucket	160	160	96	4	4	1	:	252 126
tcp_bind_bucket	339	339	32	3	3	1	:	252 126
tcp_open_request	118	118	64	2	2	1	:	252 126
inet_peer_cache	177	177	64	3	3	1	:	252 126
ip_fib_hash	15	226	32	2	2	1	:	252 126
ip_dst_cache	286	288	160	12	12	1	:	252 126
arp_cache	150	150	128	5	5	1	:	252 126
uhci_urb_priv	1	67	56	1	1	1	:	252 126
skbuff_head_cache	426	552	160	23	23	1	:	252 126
sock	165	387	832	36	43	2	:	124 62

Buffer cache initialization

The networking system is designed so that all protocol families use a common buffer structure. This structure, sometimes called a *socket buffer*, or *sk_buff* consists of at least two independent elements

- a fixed size header of type *struct sk_buff*

- a *character array* that holds the actual packet data (plus some other stuff control information)

A buffer is said to be cloned when two or more *sk_buff* headers point to the same data. In the *fclone cache* the size of a basic element is *two buffer headers plus an atomic_t following the second buffer*. This *atomic_t* entity is a *variable with no name*. Buffers are allocated from this cache when cloning is virtually assured.

[zNET]: Implement SKB fast cloning. Protocols that make extensive use of SKB cloning, for example TCP, eat at least 2 allocations per packet sent as a result. To cut the *kmalloc()* count in half, we implement a pre-allocation scheme wherein we allocate 2 *sk_buff* objects in advance, then use a simple reference count to free up the memory at the correct time. Based upon an initial patch by Thomas Graf and suggestions from Herbert Xu.

```
2397 void __init skb_init(void)
2398 {
2399     skbuff_head_cache = kmem_cache_create("skbuff_head_cache",
2400                                           sizeof(struct sk_buff),
2401                                           0,
2402                                           SLAB_HWCACHE_ALIGN|SLAB_PANIC,
2403                                           NULL);
2404
2405     skbuff_fclone_cache = kmem_cache_create("skbuff_fclone_cache",
2406                                           (2*sizeof(struct sk_buff)) +
2407                                           sizeof(atomic_t),
2408                                           0,
2409                                           SLAB_HWCACHE_ALIGN|SLAB_PANIC,
2410                                           NULL);
2411 }
```

Sock file system initialization

The *sock filesystem* is a virtual file system *VFS* in which the *inode* associated with each open socket resides. Here the system is registered and mounted.

```
305 static struct file_system_type sock_fs_type = {
306     .name = "sockfs",
307     .get_sb = sockfs_get_sb,
308     .kill_sb = kill_anon_super,
309 };
```

Since sockets are accessed through the file system via *read()* and *write()* functions, it is necessary to represent them within the Linux file system. To that end, a virtual file system is created here. This file system will contain the *inode* associated with each *socket* created by an application. The *socket* structure used to be *embedded within the inode structure*.

Now both structures are mapped in a common container.

```
682 struct socket_alloc {
683     struct socket socket;
684     struct inode vfs_inode;
685 };
```

cat /proc/filesystems may be used to verify the presence of the *sockfs* filesystem.

```
[root@stephen /proc]# cat filesystems
nodev    sysfs
nodev    rootfs
nodev    bdev
nodev    proc
nodev    cpuset
nodev    binfmt_misc
nodev    debugfs
nodev    securityfs
nodev    sockfs
nodev    usbfs
nodev    pipefs
nodev    futexfs
nodev    tmpfs
nodev    inotifyfs
nodev    eventpollfs
nodev    devpts
        ext2
nodev    ramfs
nodev    hugetlbfs
        iso9660
nodev    mqueue
nodev    selinuxfs
        ext3
```

Protocol Initialization

The *inet_init()* function which is defined in *net/ipv4/af_inet.c* is responsible for registering several data structures that bind different layers of the protocol implementation together. It is added to the *initcall* table via the following line in *af_inet.c*.

```
1355 fs_initcall(inet_init);
```

The structures involved in defining protocol layers and their connections include:

<i>struct net_proto_family</i>	Defines the protocol family's handler for socket creation. Unlike the structures below which apply specifically to IP protocols, this one is generic.
<i>struct net_protocol</i>	Specifies an IP transport protocol's receive handler for upcalls from the IP Layer.
<i>struct inet_protosw</i>	Defines a transport protocol. This structure contains pointers to instances of the two structures that follow:
<i>struct proto_ops</i>	Defines the downcall binding between the socket and AF_inet layers.
<i>struct proto</i>	Defines the downcall binding between the AF_inet layer and the true transport protocol.
<i>struct packet_type</i>	Defines upcall binding between <i>dev</i> and <i>network</i> layers. This one is also generic.

Registration of core networking data structures:

- When a new *protocol family* is initialized, it must call *sock_register()* to register its handler for socket creation.
- When a *network layer protocol* is initialized, it must call *dev_add_pack()* to register its upcall handler for packets being delivered by the *dev* layer.
- For a *transport protocol* to become operational, it must call the kernel functions shown in blue.
 - The call to *proto_register()* must come first and is used to create the cache of instances of *struct sock*
 - *inet_register_protosw()* creates the down call bindings between socket and AF_INET and AF_INET and transport protocol functions.
 - *net_add_protocol()* creates the up call binding between the network layer and the transport protocols receive entry point.

Function	structure	instance
<i>proto_register()</i>	<i>proto</i>	<i>tcp_prot, udp_prot, cop_prot</i>
<i>sock_register()</i>	<i>net_proto_family</i>	<i>inet_family_ops</i>
<i>net_add_protocol()</i>	<i>net_protocol</i>	<i>tcp_protocol, udp_protocol</i>
<i>inet_register_protosw()</i>	<i>inet_protosw</i>	<i>inetsw_array[i], cop_protosw</i>
<i>dev_add_pack()</i>	<i>packet_type</i>	<i>ip_packet_type</i>

Residence and use of core networking data structures

The leftmost column of the following table contains the name of the kernel table or pointer that serves as a base for accessing the above structures during normal operation.

Table	structure	lookup	keys	use
inet_protos	net_protocol	hash	protocol (TCP=6)	input delivery to transpt layer
inetsw	inet_protosw	hash/scan	sock type, protocol	binding to socket API functions
ptype_base	packet_type	hash/scan	eth_proto (IP=0x800)	input deliver to network layer
net_families	net_proto_family	hash	AF (AF_INET=2)	creation of a new socket

The *inet_init()* function

The first order of business is to call *proto_register()* to allocate the cache of struct sock elements that will be used by the transport protocols. The second parameter's value of 1 tells *proto_register()* that it must allocate the *struct sock* cache.

```
1416 static int __init inet_init(void)
1417 {
1418     struct sk_buff *dummy_skb;
1419     struct inet_protosw *q;
1420     struct list_head *r;
1421     int rc = -EINVAL;
1422
1423     BUILD_BUG_ON(sizeof(struct inet_skb_parm) >
1424                  sizeof(dummy_skb->cb));
1425
1426     rc = proto_register(&tcp_prot, 1);
1427     if (rc)
1428         goto out;
1429
1430     rc = proto_register(&udp_prot, 1);
1431     if (rc)
1432         goto out_unregister_tcp_proto;
1433
1434     rc = proto_register(&raw_prot, 1);
1435     if (rc)
1436         goto out_unregister_udp_proto;
```

The pointers passed to *proto_register()*, are the addresses of the *struct proto* data structures that define the downcall bindings between *af_inet()* and the true transport protocols.

The *proto_register()* function

The main job of *proto_register* is to create a cache of protocol specific *struct sock* structures. The organization is currently somewhat baroque. A pointer to any of the three structures shown below can be used as a pointer to any other one.

```
struct sock sk
{
    /* generic sock stuff */
};

struct inet_sock
{
    struct sock sk;          /* must come first */
    /* Inet dependent stuff here */
};

struct udp_sock
{
    struct inet_sock inet;  /* Must come first */
    /* Udp dependent stuff here */
};
```

The *struct cop_sock* -

You will build a *struct cop_sock* in the following way. State information and performance counters will comprise the "COP dependent stuff."

```
struct cop_sock
{
    struct inet_sock inet;  /* Must come first */
    /* COP dependent stuff here */
};
```

The `proto_register()` function

The use of the *request sock cache* (*rsk*) and the *timed wait* (*twsk*) caches are not well understood at present. They are *not used* in simple transport protocols. The *obj_size* element of the *struct proto* must be initialized to the correct size *before* calling `proto_register()`.

The *obj_size* is the size of the composite sock (e.g. `sizeof(struct udp_sock)` or `sizeof(struct cop_sock)`).

```
2036 int proto_register(struct proto *prot, int alloc_slab)
2037 {
2038     if (alloc_slab) {
2039         prot->slab = kmem_cache_create(prot->name,
2040                                     prot->obj_size, 0, SLAB_HWCACHE_ALIGN, NULL);
2041
2042         if (prot->slab == NULL) {
2043             printk(KERN_CRIT "%s: Can't create sock SLAB
2044 cache!\n",
2045                 prot->name);
2046             goto out;
2047         }
2048         if (prot->rsk_prot != NULL) {
2049             static const char mask[] = "request_sock_%s";
2050             :
2051         }
2052         if (prot->twsk_prot != NULL) {
2053             static const char mask[] = "tw_sock_%s";
2054             :
2055         }
2056         write_lock(&proto_list_lock);
2057         list_add(&prot->node, &proto_list);
2058         assign_proto_idx(prot);
2059         write_unlock(&proto_list_lock);
2060         return 0;
2061     }
2062 }
```

Registering the protocol family

The *inet_init()* function continues by calling *sock_register()*. This creates the binding between AF_INET and the *inet_create()* function.

```
1437     /*
1438     *       Tell SOCKET that we are alive...
1439     */
1440
1441     (void)sock_register(&inet_family_ops);
1442
1443 #ifdef CONFIG_SYSCTL
1444     ip_static_sysctl_init();
1445 #endif
```

Protocol family data structures

For a network protocol family (PF_INET, PF_IPX, PF_APPLETALK, etc..) to register and bind itself to the socket layer *it must create a structure of the type net_proto_family*.

The *net_proto_family* data structure describes each of the supported *protocol families (protocol stacks, network architectures)*.

```
192 struct net_proto_family {
193     int      family;
194     int      (*create)(struct net *net, struct socket *sock,
195                       int protocol);
195     struct module *owner;
196 };
```

Two elements of this structure are especially important:

The **address family (AF_)** is a unique identifier associated with each protocol stack supported by Linux.

The **create function pointer** is used to register a protocol handler function that is to be called whenever a new socket of this protocol type is created.

A transport protocol such as COP that is part of an existing family (PF_INET) must never call `sock_register`.

The *net_proto_family* structure for IPV4.

This *net_proto_family* structure for IPV4 is statically declared in [linux/net/ipv4/af_inet.c](#) and initialized as shown:

```
921 static struct net_proto_family inet_family_ops = {
922     .family = PF_INET,
923     .create = inet_create,
924     .owner  = THIS_MODULE,
925 };
```

The *create* entry in the structure is a pointer to the function that is to be called by the socket layer any time an AF_INET socket is created:

e.g.

```
socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

results in a call to *inet_create()* which actually controls the creation of the new socket.

The `sock_register()` function

The `sock_register()` function is defined in `net/socket.c`. Its mission is to store a pointer to the `net_proto_family` into the static `net_families` address table.

The protocol number (`AF_INET`, `AF_IPX`, `AF_APPLETALK`) serves as the index.

```
134 static struct net_proto_family *net_families[NPROTO];
```

The size of the `net_proto_family` table is determined by the `NPROTO` which is defined in `include/linux/net.h` and defines the maximum number of supported protocol stacks.

```
27 #define NPROTO          32 /* should be enough for now.. */
```

It makes sure the family number is legitimate and then, if the *protocol is not already registered*, it stores a pointer to the `ops` structure in the `net_families` table indexed by `ops->family`.

```
2160 int sock_register(const struct net_proto_family *ops)
2161 {
2162     int err;
2163
2164     if (ops->family >= NPROTO) {
2165         printk(KERN_CRIT "protocol %d >= NPROTO(%d)\n", ops->family,
2166                 NPROTO);
2167         return -ENOBUFS;
2168     }
2169
2170     spin_lock(&net_family_lock);
2171     if (net_families[ops->family])
2172         err = -EEXIST;
2173     else {
2174         net_families[ops->family] = ops;
2175         err = 0;
2176     }
2177     spin_unlock(&net_family_lock);
2178
2179     printk(KERN_INFO "NET: Registered protocol family %d\n",
2180            ops->family);
2180     return err;
2181 }
```

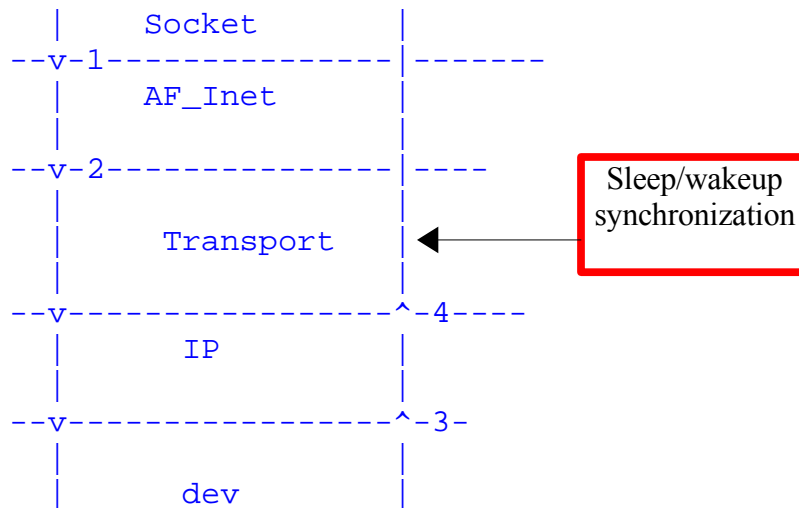
Registering the *Transport* protocols

Upon return to *inet_init* the next order of business is to construct a table of handlers for the IP transport protocols. A rather complicated binding scheme is in play here.

A transport layer protocol must connect itself both to the IP layer below it and also to the AF_INET layer above it. Since the protocols are duplex, the bindings might more properly be considered as consisting of down-bindings shown on the left and up-bindings shown on the right.

In addition to the bindings discussed here, static bindings are used for transport to IP down-calls and sleep/wakeup synchronization is used at the point at which data is passed between downward receive calls and the softirq upcalls triggered by packet reception.

Note that downcalls are used on the receive path as well as the send path at locations 1 and 2. The final upcall in the upcall path occurs at location 4. The remainder of receive occurs via *return* from the downcall. The unlabeled nodes on the downcall side represent *ad hoc* bindings. The missing nodes on the receive side represent returns from downcalls.



Downcalls

- 1 - struct proto_ops
- 2 - struct proto

Upcalls

- 3 - struct packet_type
- 4 - struct inet_protocol

The IP to Transport Receive (upcall) Binding

We begin with a discussion of the receive (upcall) binding from the IP layer (point 4 in the diagram) to the transport layer. Each of the supported transports must register a transport descriptor structure called the *net_protocol* defined in *include/net/protocol.h*. This structure was formerly called the *inet_protocol*.

The *gso_* prefix stands for *generic segmentation offload*. Many modern NICs support TSO (TCP segmentation offload) in which the stack can pass TCP segments of up to 64KB in size directly to the NIC. The NIC is responsible for resegmenting into MTU size packets --- This is NOT IP FRAGMENTATION!! GSO is a new *feature* in which other protocols can pass large segments down the stack.

```
35 /* This is used to register protocols. */
36 struct net_protocol {
37     int      (*handler)(struct sk_buff *skb);
38     void     (*err_handler)(struct sk_buff *skb, u32 info);
39     int      (*gso_send_check)(struct sk_buff *skb);
40     struct sk_buff *(*gso_segment)(struct sk_buff *skb,
41                                   int features);
42     unsigned int  no_policy:1,
43                 netns_ok:1;
44 };
```

The functions of individual structure elements are described below:

handler: This is a pointer to the packet reception entry point of the transport protocol. That is, when the IP layer receives a packet and determines from the protocol byte in the header that the packet is destined for *this* transport, it will invoke the handler function passing it a pointer to the *sk_buff* which holds the packet.

err_handler: Similar to handler but used for processing error events.

Static declarations of the IP to transport receive bindings

The standard IP transport protocol descriptors are statically defined in *net/ipv4/protocol.c*

Note that UDP and ICMP *don't do gso*. We won't either.

```
1317 static struct net_protocol tcp_protocol = {
1318     .handler =      tcp_v4_rcv,
1319     .err_handler =  tcp_v4_err,
1320     .gso_send_check = tcp_v4_gso_send_check,
1321     .gso_segment =  tcp_tso_segment,
1322     .no_policy =    1,
1323     .netns_ok =     1,
1324 };
1325
1326 static struct net_protocol udp_protocol = {
1327     .handler =      udp_rcv,
1328     .err_handler =  udp_err,
1329     .no_policy =    1,
1330     .netns_ok =     1,
1331 };
1332
1333 static struct net_protocol icmp_protocol = {
1334     .handler =      icmp_rcv,
1335     .no_policy =    1,
1336     .netns_ok =     1,
1337 };
```

Registering the upcall bindings

Back in *inet_init()* the upcall bindings for the transport protocols are added.

```
1447     /*
1448     *       Add all the base protocols.
1449     */
1450
1451     if (inet_add_protocol(&icmp_protocol, IPPROTO_ICMP) < 0)
1452         printk(KERN_CRIT "inet_init: Cannot add ICMP protocol\n");
1453     if (inet_add_protocol(&udp_protocol, IPPROTO_UDP) < 0)
1454         printk(KERN_CRIT "inet_init: Cannot add UDP protocol\n");
1455     if (inet_add_protocol(&tcp_protocol, IPPROTO_TCP) < 0)
1456         printk(KERN_CRIT "inet_init: Cannot add TCP protocol\n");
1457 #ifdef CONFIG_IP_MULTICAST
1458     if (inet_add_protocol(&igmp_protocol, IPPROTO_IGMP) < 0)
1459         printk(KERN_CRIT "inet_init: Cannot add IGMP protocol\n");
1460 #endif
1461
```

Adding a single transport layer input handler

The mechanics of the hash queue construction are performed by *inet_add_protocol()*. The hash function *used to* use the least significant 5 bits of the protocol number, and the queuing logic *inserted new entries at the head of the hash chain*.

Now MAX_INET_PROTOS is 255 and only one handler per protocol is allowed.

```
52
53 int inet_add_protocol(struct net_protocol *prot,
                        unsigned char protocol)
54 {
55     int hash, ret;
56
57     hash = protocol & (MAX_INET_PROTOS - 1);
58
59     spin_lock_bh(&inet_proto_lock);
60     if (inet_protos[hash]) {
61         ret = -1;
62     } else {
63         inet_protos[hash] = prot;
64         ret = 0;
65     }
66     spin_unlock_bh(&inet_proto_lock);
67
68     return ret;
69 }
70
```

The old method

The Linux implementation used to support multiple handlers for a given transport protocol. The hash chain of protocols is searched to see if a registered handler already exists. The copy bit was set whenever a duplicate handler is encountered. Presumably the IP layer cloned the skb and passed it to both.

```
119 /*
120 *      Set the copy bit if we need to.
121 */
122
123     p2 = (struct inet_protocol *)prot->next;
124     while (p2) {
125         if (p2->protocol == prot->protocol) {
126             prot->copy = 1;
127             break;
128         }
129         p2 = (struct inet_protocol *) p2->next;
130     }
131     br_write_unlock_bh(BR_NETPROTO_LOCK);
132 }
```

This completes the up binding of the IP layer to the particular transport.

Down call bindings to the socket layer

As shown in the figure, the socket layer must be bound to the AF_inet layer and the AF_inet layer must be bound to the transport layer. Both of these bindings are effected via the *struct inet_protosw* structure defined in *include/net/protocol.h*. The general form of the lookup key that will be used to identify the protocol when a socket is created consists of *both* the socket type and the transport protocol number (e.g. *fd = socket(AF_INET, SOCK_NTP, IPPROTO_NTP);*)

```
70 struct inet_protosw {
71     struct list_head list;
72
73     /* These two fields form the lookup key. */
74     unsigned short type; /* 2nd argument to socket(2). */
75     unsigned short protocol; /* This is the L4 protocol num */
76
77     struct proto *prot;
78     const struct proto_ops *ops;
79
80     int capability; /* Which (if any) capability do
81                    * we need to use this socket
82                    * interface?
83                    */
84     char no_check; /* checksum on rcv/xmit/none? */
85     unsigned char flags; /* See INET_PROTOSW_* below. */
86 };
87 #define INET_PROTOSW_REUSE 0x01 /* Ports automatically reusable? */
88 #define INET_PROTOSW_PERMANENT 0x02 /* Permanent protocols are
89                                     unremovable. */
89 #define INET_PROTOSW_ICSK 0x04 /* Is this an
89                                     inet_connection_sock? */
```

Functions of structure elements:

- list:* Used to link these structures together. Each list is based in the *inetsw[]* array as *indexed by socket type* shown below.
- type:* SOCK_STREAM, SOCK_DGRAM, SOCK_RAW
- protocol:* The standard protocol number for the transport IPPROTO_TCP (6), IPPROTO_UDP(17). IPPROTO_IP(0) is used as a *wild card*.

The *prot* and *ops* elements are pointers to additional structures used to establish downcall bindings.

ops: This *struct proto_ops* table of function pointers comprises the downcall binding of the **socket layer to the AF_INET layer**. It is a table of pointers to AF_INET's table of primary entry points to the handlers for the standard repertoire of socket operations (e.g. bind, accept, sendto, recvfrom). The functions themselves are found in *net/ipv4/af_inet.c*. There are only two sets of these function pointers associated with TCP/IPV4: *inet_dgram_ops* which is used by UDP, and *inet_stream_ops* used by TCP.

prot: This *struct proto* table of function pointers comprises the downcall binding of the **AF_inet layer to the transport layer**. These are pointers to the transport specific handlers for the generic socket operations. That is, they point to functions in *tcp*.c* *udp*.c*, etc. There are only three sets of these functions in use: *tcp_prot*, *udp_prot*, and *raw_prot*.

The *inet_protosw* array

The structures used in TCP/IP are statically defined in *net/ipv4/af_inet.c*. Recall that the socket system call has three parameters:

```
int socket(int protocol_family, int socket_type, int protocol_id);
```

Here we are considering the initialization of IPV4 and so the domain is necessarily PF_INET. Thus there is **one instance in the *inetsw_array[]* for each legitimate combination of (*type, protocol*)**.

```
930 static struct inet_protosw inetsw_array[] =
931 {
932     {
933         .type =      SOCK_STREAM,    /* 1 */
934         .protocol =  IPPROTO_TCP,    /* 6 */
935         .prot =      &tcp_prot,
936         .ops =       &inet_stream_ops,
937         .capability = -1,
938         .no_check =  0,
939         .flags =     INET_PROTOSW_PERMANENT |
940                   INET_PROTOSW_ICSK,
941     },
942     {
943         .type =      SOCK_DGRAM,     /* 2 */
944         .protocol =  IPPROTO_UDP,    /* 17 */
945         .prot =      &udp_prot,
946         .ops =       &inet_dgram_ops,
947         .capability = -1,
948         .no_check =  UDP_CSUM_DEFAULT,
949         .flags =     INET_PROTOSW_PERMANENT,
950     },
951 },
952     {
953         .type =      SOCK_RAW,       /* 3 */
954         .protocol =  IPPROTO_IP,     /* wild card */
955         .prot =      &raw_prot,
956         .ops =       &inet_sockraw_ops,
957         .capability = CAP_NET_RAW,
958         .no_check =  UDP_CSUM_DEFAULT,
959         .flags =     INET_PROTOSW_REUSE,
960     }
961 }
962 };
```

Socket type values:

The supported values of *type* are defined in *include/asm/socket.h*

```
51 #define SOCK_STREAM      1 /* stream (connection) socket */
52 #define SOCK_DGRAM      2 /* datagram (conn.less) socket */
53 #define SOCK_RAW        3 /* raw socket */
54 #define SOCK_RDM        4 /* reliably-delivered message */
55 #define SOCK_SEQPACKET  5 /* sequential packet socket */
56 #define SOCK_PACKET     10
61 #define SOCK_MAX (SOCK_PACKET+1)
```

The UDP *struct proto*

The use of *struct proto* and *struct proto_ops* in the context of UDP is illustrated below. The table *udp_prot* contains pointers to UDP specific handlers that are invoked as needed by AF_inet layer. You will need to build one of these structures for your protocol. To register your protocol, the only element that you *must* initialize is the *obj_size*! To actually send or receive anything you will need to provide *sendmsg* and *recvmsg* handlers.

```
1472 struct proto udp_prot = {
1473     .name           = "UDP",
1474     .owner          = THIS_MODULE,
1475     .close          = udp_lib_close,
1476     .connect        = ip4_datagram_connect,
1477     .disconnect     = udp_disconnect,
1478     .ioctl          = udp_ioctl,
1479     .destroy        = udp_destroy_sock,
1480     .setsockopt     = udp_setsockopt,
1481     .getsockopt     = udp_getsockopt,
1482     .sendmsg        = udp_sendmsg,
1483     .recvmsg        = udp_recvmsg,
1484     .sendpage       = udp_sendpage,
1485     .backlog_rcv    = __udp_queue_rcv_skb,
1486     .hash           = udp_lib_hash,
1487     .unhash         = udp_lib_unhash,
1488     .get_port       = udp_v4_get_port,
1489     .memory_allocated = &udp_memory_allocated,
1490     .sysctl_mem     = sysctl_udp_mem,
1491     .sysctl_wmem    = &sysctl_udp_wmem_min,
1492     .sysctl_rmem    = &sysctl_udp_rmem_min,
1493     .obj_size       = sizeof(struct udp_sock),
1494     .h.udp_hash     = udp_hash,
1495 #ifdef CONFIG_COMPAT
1496     .compat_setsockopt = compat_udp_setsockopt,
1497     .compat_getsockopt = compat_udp_getsockopt,
1498 #endif
1499 };
```

The `inet_dgram struct proto_ops`

The `proto_ops` tables contain functions exported by the AF_INET layer to the socket layer. The functions which start with `sock_no` are not implemented and thus point to default handlers in the socket layer above. Both UDP and TCP rely upon some common AF_INET layer functions.

Since TCP is connection oriented and UDP is not, UDP does not implement `accept` and `listen` but TCP must. You will need to supply one of these structures for your protocol. You should either statically or dynamically copy this one.

```
867 const struct proto_ops inet_dgram_ops = {
868     .family           = PF_INET,
869     .owner            = THIS_MODULE,
870     .release          = inet_release,
871     .bind             = inet_bind,
872     .connect          = inet_dgram_connect,
873     .socketpair       = sock_no_socketpair,
874     .accept           = sock_no_accept,
875     .getname          = inet_getname,
876     .poll             = udp_poll,
877     .ioctl            = inet_ioctl,
878     .listen           = sock_no_listen,
879     .shutdown         = inet_shutdown,
880     .setsockopt       = sock_common_setsockopt,
881     .getsockopt       = sock_common_getsockopt,
882     .sendmsg          = inet_sendmsg,
883     .recvmsg          = sock_common_recvmsg,
884     .mmap             = sock_no_mmap,
885     .sendpage         = inet_sendpage,
886 #ifdef CONFIG_COMPAT
887     .compat_setsockopt = compat_sock_common_setsockopt,
888     .compat_getsockopt = compat_sock_common_getsockopt,
889 #endif
890 }
```

The `inet_stream struct proto_ops`

Because the stream protocols are designed to support connection orientation the *accept* and *listen* functions are no longer null.

```
841 const struct proto_ops inet_stream_ops = {
842     .family           = PF_INET,
843     .owner            = THIS_MODULE,
844     .release          = inet_release,
845     .bind             = inet_bind,
846     .connect          = inet_stream_connect,
847     .socketpair       = sock_no_socketpair,
848     .accept           = inet_accept,
849     .getname          = inet_getname,
850     .poll             = tcp_poll,
851     .ioctl            = inet_ioctl,
852     .listen           = inet_listen,
853     .shutdown         = inet_shutdown,
854     .setsockopt       = sock_common_setsockopt,
855     .getsockopt       = sock_common_getsockopt,
856     .sendmsg          = tcp_sendmsg,
857     .recvmsg          = sock_common_recvmsg,
858     .mmap             = sock_no_mmap,
859     .sendpage         = tcp_sendpage,
860     .splice_read      = tcp_splice_read,
861 #ifdef CONFIG_COMPAT
862     .compat_setsockopt = compat_sock_common_setsockopt,
863     .compat_getsockopt = compat_sock_common_getsockopt,
864 #endif
865 };
```

Registering the standard transports in *inet_init()*

To facilitate speedy access to the required *struct inet_protosw* structure during socket creation the structures are actually accessed via an array of list headers using the socket type (e.g. *SOCK_DGRAM*) as an index into the list. The array of list headers resides in *net/ipv4/af_inet.c*.

```
150 /* The inetsw table contains everything that inet_create needs
151 *   to build a new socket.
152 */
153     struct list_head inetsw[SOCK_MAX];
154
```

The list headers for each of the possible socket types are initialized by *inet_init()*

```
1141 /* Register the socket-side information for inet_create. */
1142     for(r = &inetsw[0]; r < &inetsw[SOCK_MAX]; ++r)
1143         INIT_LIST_HEAD(r);
```

After the list headers are initialized, each element in the statically created table of structures of type *inet_protosw* is inserted into the appropriate list via a call to *inet_register_protosw()*. **Your module will need to invoke *inet_register_protosw()*.**

```
1466     for (q = inetsw_array; q < &inetsw_array[INETSW_ARRAY_LEN];
1467           ++q)
1467         inet_register_protosw(q);
1468
```

inet_init() completes by calling initialization entry points of the remaining elements of the TCP/IP stack that require initialization.

```
1473     arp_init();
1474
1475     /*
1476      *       Set the IP module up
1477      */
1478
1479     ip_init();
1480
1481     tcp_v4_init();
1482
1483     /* Setup TCP slab cache for open requests. */
1484     tcp_init();
1485
1486     /* Setup UDP memory threshold */
1487     udp_init();
1488
1489     /* Add UDP-Lite (RFC 3828) */
1490     udplite4_register();
1491
1492     if (icmp_init() < 0)
1493         panic("Failed to create the ICMP control socket.\n");
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513     ipv4_proc_init();
1514
1515     ipfrag_init();
1516
1517     dev_add_pack(&ip_packet_type);
1518
1519     rc = 0;
1520 out:
1521     return rc;
1522
1523
1524
1525
1526
1527 }
1528 }
1529 fs_initcall(inet_init);
1530
```

Registering *inet_protosw* structures

The *inet_register_protosw()* function, also defined in `net/ipv4/af_inet.c`, is responsible for inserting each *inet_proto_sw* structure into the list of such structures based at *inetsw[p->type]* (where type = SOCK_STREAM, .. etc). This routine is called not only by *init_inet()*, but *must also be called when any private transport implemented as a module is loaded*. The individual *inetsw[]* lists consist of a single element unless there are multiple transports associated with a

```
967 void inet_register_protosw(struct inet_protosw *p)
968 {
969     struct list_head *lh;
970     struct inet_protosw *answer;
971     int protocol = p->protocol;
972     struct list_head *last_perm;
973
974     spin_lock_bh(&inetsw_lock);
975
976     if (p->type >= SOCK_MAX)
977         goto out_illegal;
978
```

The list of all *inet_protosw*'s having the same type as the one being registered is searched. If an existing protocol number (TCP(6), UDP(17), ..etc) matches the one being registered, *and* the existing protocol is flagged *PERMANENT* the new protocol can't be registered. The entries for both UDP and TCP have the *INET_PROTOSW_PERMANENT* attribute.

```
979 /* If we are trying to override a permanent protocol, bail. */
980     answer = NULL;
981     last_perm = &inet_sw[p->type];
982     list_for_each(lh, &inet_sw[p->type]) {
983         answer = list_entry(lh, struct inet_protosw, list);
984
985         /* Check only the non-wild match. */
986         if (INET_PROTOSW_PERMANENT & answer->flags) {
987             if (protocol == answer->protocol)
988                 break;
989             last_perm = lh;
990         }
991
992         answer = NULL;
993     }
994     if (answer)
995         goto out_permanent;
996
```

If it is legal to add this protocol, then it is added to the list after the last permanent protocol. This has the effect of overriding any existing non-permanent entry, and if this entry is removed, the system automatically returns to the old behavior. **Given that this is the case, why does it make sense to be able to register multiple input handlers for a given protocol??** Since that's no longer allowed, I guess they saw the light!

```

 997 /* Add the new entry after the last permanent entry if any, so that
 998 * the new entry does not override a permanent entry when matched with
 999 * a wild-card protocol. But it is allowed to override any existing
1000 * non-permanent entry. This means that when we remove this entry,
1001 * the system automatically returns to the old behavior.
1002 */
1003     list_add_rcu(&p->list, last_perm);
1004 out:
1005     spin_unlock_bh(&inetsw_lock);
1006
1007     synchronize_net();
1008
1009     return;
1010
1011 out_permanent:
1012     printk(KERN_ERR "Attempt to override permanent protocol %d.\n",
1013            protocol);
1014     goto out;
1015
1016 out_illegal:
1017     printk(KERN_ERR
1018            "Ignoring attempt to register invalid socket type %d.\n",
1019            p->type);
1020     goto out;
1021 }
```

Binding the receive side of the IP stack to the *dev* layer

Two "standard" logical link level protocols (LLC's) have been used in the Ethernet

RFC 894 (a.k.a. DIX = Digital / Intel / Xerox)
 IEEE 802.2 & IEEE 802.3 (RFC 1042)

RFC 894

Dest addr	Source addr	Packet type	Data	CRC
6	6	2	46-1500	4

Packet types include

0x0800 IP Datagram
 0x0806 ARP Request / Reply
 0x8035 RARP Request / Reply

RFC 1042

<---- 802.2 LLC ----><-802.2 SNAP->

Dest Addr	Source Addr	Length	DSAP 0xaa	SSAP 0xaa	Cntl 0x03	Org Code 0x00.	Type	Data	CRC
6	6	2	1	1	1	3	2	38-1492	4

This, alas, provides multiple three ways in which incoming packets must be demuxed at the link layer:

RFC 894 Packet type *dev_add_pack()*
 RFC 1042 Packet type *register_snap_client()*
 IEEE 802.2 LLC DSAP *register_8022_client()*

Exercise: *What happens if some host system encapsulates IP data using 802.2 but the linux system has not registered an 802.2 client??*

Registering the IP Packet type

The *struct packet_type* is used by a network layer protocol to register its input handler with the *dev* layer.

```
892 struct packet_type {
893     __be16  type; /* This is really htons(ether_type). */
894     struct net_device *dev; /* NULL is wildcarded here */
895     int      (*func) (struct sk_buff *,
896                     struct net_device *,
897                     struct packet_type *,
898                     struct net_device *);
899     struct sk_buff *(*gso_segment)(struct sk_buff *skb,
900                                  int features);
901     int      (*gso_send_check)(struct sk_buff *skb);
902     void      *af_packet_priv;
903     struct list_head list;
904};
```

Functions of structure members:

type: One of the packet types specified in `/include/linux/if_ether.h`. (But stored in network byte order!). These are industry wide link layer protocol identifiers.

```
42 #define ETH_P_IP      0x0800
43 #define ETH_P_X25     0x0805
44 #define ETH_P_ARP     0x0806
```

dev: Device from which to receive packets. NULL is a wild card meaning packets may be received from all devices.

func: This is a pointer to the packet reception entry point of the protocol. When a packet is received by the link layer, it is routed to this handler if the *type* field in the MAC header matches the type specified above.

The IP specific instance of the structure is statically defined as:

```
1409 static struct packet_type ip_packet_type = {
1410     .type = __constant_htons(ETH_P_IP),
1411     .func = ip_rcv,
1412     .gso_send_check = inet_gso_send_check,
1413     .gso_segment = inet_gso_segment,
1414 };
```

- The *ip_rcv()* function is the first level entry point for IPv4 input packet handling function.
- The handlers for specific frame types are stored in a *16 slot hash table*.
- A type value of ETH_P_ALL (instead of ETH_P_IP) would indicate the handler wants to see *all* incoming frames regardless of ethertype.

IP's receive handler is registered via the call to *dev_add_pack()*

The *dev_add_pack* function

These structures are maintained on a 16 way hashed list. The hash key is the low order 4 bits of the packet type. The standard technique used in managing the many hash queues is to insert new elements at the *head* of the list.

`./core/dev.c:`

```
static struct packet_type *ptype_base[16]; /* 16 way hashed list */
static struct packet_type *ptype_all; /* queue of ETH_P_ALLs */

382 void dev_add_pack(struct packet_type *pt)
383 {
384     int hash;
385
386     spin_lock_bh(&ptype_lock);
387     if (pt->type == htons(ETH_P_ALL))
388         list_add_rcu(&pt->list, &ptype_all);
389     else {
390         hash = ntohs(pt->type) & PTYPE_HASH_MASK;
391         list_add_rcu(&pt->list, &ptype_base[hash]);
392     }
393     spin_unlock_bh(&ptype_lock);
394 }
```

Residence and use of core networking data structures

Table	structure	lookup	keys	use
inet_protos	net_protocol	hash	protocol (TCP=6)	input delivery to transpt layer
inet_sw	inet_protosw	hash/scan	sock type, protocol	binding to socket API functions
ptype_base	packet_type	hash/scan	eth_proto (IP=0x800)	input deliver to network layer
net_families	net_proto_family	hash	AF (AF_INET=2)	creation of a new socket

Unregistering a transport protocol

Components of a transport protocol should be unregistered in reverse of the order in which they were registered:

```
inet_unregister_protosw(&inet_sw_cop);  
printk("cop_cleanup: Unregistered protosw \n");  
  
inet_del_protocol(&cop_protocol, IPPROTO_COP);  
printk("cop_cleanup: removed cop_protocol \n");  
  
proto_unregister(&cop_prot);  
printk("cop_cleanup: Unregistered cop_prot \n");
```