

# **The Linux Kernel API**

## **The Linux Kernel API**

This documentation is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

For more details see the file COPYING in the source distribution of Linux.

# Table of Contents

<b>1. Driver Basics.....</b>	<b>1</b>
1.1. Driver Entry and Exit points .....	1
module_init .....	1
module_exit.....	2
1.2. Atomic and pointer manipulation .....	3
atomic_read.....	3
atomic_set .....	4
atomic_add.....	5
atomic_sub .....	6
atomic_sub_and_test.....	7
atomic_inc.....	8
atomic_dec .....	9
atomic_dec_and_test.....	10
atomic_inc_and_test .....	11
atomic_add_negative.....	12
get_unaligned.....	13
put_unaligned.....	14
1.3. Delaying, scheduling, and timer routines .....	15
schedule_timeout .....	15
reparent_to_init.....	16
<b>2. Data Types .....</b>	<b>1</b>
2.1. Doubly Linked Lists .....	1
list_add.....	1
list_add_tail.....	2
list_del.....	3
list_del_init .....	4
list_empty.....	4
list_splice .....	5
list_entry .....	6
list_for_each.....	7
list_for_each_safe .....	8

list_for_each_prev.....	9
<b>3. Basic C Library Functions .....</b>	<b>11</b>
3.1. String Conversions .....	11
simple_strtoll .....	11
simple_strtoul.....	12
simple_strtol.....	13
simple_strtoull .....	14
vsnprintf .....	15
snprintf .....	16
vsprintf .....	17
sprintf .....	18
vsscanf.....	19
sscanf.....	20
3.2. String Manipulation .....	21
strcpy .....	21
strncpy .....	22
strcat.....	23
strncat.....	24
strcmp.....	25
strncmp.....	26
strchr .....	27
strrchr .....	28
strlen.....	28
strnlen.....	29
strpbrk .....	30
strtok .....	31
memset .....	32
bcopy .....	33
memcpy .....	34
memmove.....	35
memcmp.....	37
memscan .....	37
strstr.....	39

memchr .....	39
3.3. Bit Operations .....	41
set_bit.....	41
__set_bit.....	42
clear_bit.....	43
__change_bit.....	44
change_bit.....	45
test_and_set_bit.....	46
__test_and_set_bit.....	47
test_and_clear_bit .....	48
__test_and_clear_bit .....	49
test_and_change_bit.....	50
test_bit.....	51
find_first_zero_bit .....	51
find_next_zero_bit .....	52
ffz .....	53
ffs .....	54
hweight32.....	55
<b>4. Memory Management in Linux.....</b>	<b>57</b>
4.1. The Slab Cache .....	57
kmem_cache_create.....	57
kmem_cache_shrink .....	59
kmem_cache_destroy.....	59
kmem_cache_alloc.....	61
kmalloc.....	62
kmem_cache_free .....	63
kfree .....	64
<b>5. The proc filesystem .....</b>	<b>66</b>
5.1. sysctl interface .....	66
register_sysctl_table.....	66
unregister_sysctl_table.....	68
proc_dostring .....	69
proc_dointvec.....	70

proc_dointvec_minmax.....	72
proc_doulongvec_minmax.....	73
proc_doulongvec_ms_jiffies_minmax .....	75
proc_dointvec_jiffies.....	76
<b>6. The Linux VFS .....</b>	<b>79</b>
6.1. The Directory Cache .....	79
d_invalidate .....	79
d_find_alias .....	80
prune_dcache .....	81
shrink_dcache_sb.....	82
have_submounts.....	83
shrink_dcache_parent .....	84
d_alloc.....	84
d_instantiate .....	86
d_alloc_root .....	87
d_lookup .....	87
d_validate .....	89
d_delete .....	89
d_rehash .....	90
d_move.....	91
__d_path.....	92
is_subdir.....	94
find_inode_number .....	95
d_drop .....	96
d_add.....	97
dget.....	98
d_unhashed .....	99
6.2. Inode Handling.....	100
__mark_inode_dirty .....	100
write_inode_now.....	101
clear_inode.....	102
invalidate_inodes.....	103
get_empty_inode.....	104

iunique.....	105
insert_inode_hash .....	106
remove_inode_hash .....	107
iput .....	108
bmap.....	109
update_atime .....	110
make_bad_inode .....	111
is_bad_inode .....	112
6.3. Registration and Superblocks .....	113
register_filesystem .....	113
unregister_filesystem .....	114
get_super .....	115
6.4. File Locks.....	116
posix_lock_file.....	116
__get_lease.....	118
lease_get_mtime .....	118
posix_block_lock .....	119
posix_unblock_lock .....	120
lock_may_read.....	121
lock_may_write.....	123
fcntl_getlease .....	124
fcntl_setlease.....	125
sys_flock .....	126
get_locks_status .....	128
<b>7. Linux Networking .....</b>	<b>130</b>
7.1. Socket Buffer Functions.....	130
skb_queue_empty .....	130
skb_get .....	131
kfree_skb.....	131
skb_cloned .....	132
skb_shared.....	133
skb_share_check .....	134
skb_unshare.....	136

skb_peek .....	137
skb_peek_tail .....	138
skb_queue_len .....	139
__skb_queue_head .....	140
skb_queue_head .....	141
__skb_queue_tail .....	142
skb_queue_tail .....	143
__skb_dequeue .....	144
skb_dequeue .....	145
skb_insert .....	146
skb_append .....	147
skb_unlink .....	148
__skb_dequeue_tail .....	149
skb_dequeue_tail .....	150
skb_put .....	151
skb_push .....	152
skb_pull .....	153
skb_headroom .....	155
skb_tailroom .....	155
skb_reserve .....	156
skb_trim .....	157
skb_orphan .....	158
skb_queue_purge .....	159
__skb_queue_purge .....	160
__dev_alloc_skb .....	161
dev_alloc_skb .....	163
skb_cow .....	164
skb_over_panic .....	165
skb_under_panic .....	166
alloc_skb .....	167
__kfree_skb .....	168
skb_clone .....	169
skb_copy .....	170
pskb_copy .....	171



pskb_expand_head.....	172
skb_copy_expand.....	174
__pskb_pull_tail.....	175
7.2. Socket Filter .....	177
sk_run_filter .....	177
sk_chk_filter.....	178
<b>8. Network device support.....</b>	<b>180</b>
8.1. Driver Support.....	180
init_etherdev .....	180
alloc_etherdev .....	181
init_fddidev .....	182
alloc_fddidev.....	183
init_hippi_dev .....	184
alloc_hippi_dev .....	185
init_trdev .....	186
alloc_trdev.....	187
init_fcdev .....	188
alloc_fcdev .....	190
dev_add_pack .....	191
dev_remove_pack .....	192
__dev_get_by_name .....	193
dev_get_by_name .....	194
dev_get.....	195
__dev_get_by_index .....	196
dev_get_by_index .....	197
dev_alloc_name .....	198
dev_alloc .....	199
netdev_state_change .....	200
dev_load.....	201
dev_open .....	202
dev_close.....	203
register_netdevice_notifier.....	204
unregister_netdevice_notifier.....	205

dev_queue_xmit.....	206
netif_rx.....	207
net_call_rx_atomic.....	208
register_gifconf.....	209
netdev_set_master.....	210
dev_set_promiscuity.....	211
dev_set_allmulti.....	213
dev_ioctl.....	214
dev_new_index.....	215
netdev_finish_unregister.....	216
unregister_netdevice.....	217
8.2. 8390 Based Network Cards.....	218
ei_open.....	218
ei_close.....	219
ei_tx_timeout.....	220
ei_interrupt.....	220
ethdev_init.....	222
NS8390_init.....	223
8.3. Synchronous PPP.....	224
sppp_input.....	224
sppp_close.....	225
sppp_open.....	226
sppp_reopen.....	227
sppp_change_mtu.....	228
sppp_do_ioctl.....	229
sppp_attach.....	230
sppp_detach.....	231
<b>9. Module Support .....</b>	<b>233</b>
9.1. Module Loading.....	233
request_module.....	233
call_usermodehelper.....	234
9.2. Inter Module support.....	235
inter_module_register.....	235

inter_module_unregister .....	236
inter_module_get .....	237
inter_module_get_request .....	238
inter_module_put .....	239
<b>10. Hardware Interfaces .....</b>	<b>241</b>
10.1. Interrupt Handling .....	241
disable_irq_nosync .....	241
disable_irq .....	242
enable_irq .....	243
probe_irq_mask .....	244
10.2. MTRR Handling .....	245
mtrr_add .....	245
mtrr_del .....	247
10.3. PCI Support Library .....	248
pci_find_slot .....	248
pci_find_subsys .....	249
pci_find_device .....	251
pci_find_class .....	252
pci_find_capability .....	253
pci_find_parent_resource .....	255
pci_set_power_state .....	256
pci_save_state .....	257
pci_restore_state .....	258
pci_enable_device .....	259
pci_disable_device .....	260
pci_enable_wake .....	261
pci_release_regions .....	262
pci_request_regions .....	263
pci_match_device .....	264
pci_register_driver .....	265
pci_unregister_driver .....	266
pci_announce_device_to_drivers .....	267
pci_insert_device .....	268

pci_remove_device .....	269
pci_dev_driver.....	270
pci_set_master.....	271
pci_setup_device .....	272
pci_pool_create .....	273
pci_pool_destroy .....	275
pci_pool_alloc.....	275
pci_pool_free .....	277
10.4. MCA Architecture .....	278
10.4.1. MCA Device Functions .....	278
mca_find_adapter.....	278
mca_find_unused_adapter.....	279
mca_read_stored_pos.....	280
mca_read_pos .....	281
mca_write_pos .....	282
mca_set_adapter_name .....	284
mca_set_adapter_procfn .....	285
mca_is_adapter_used .....	286
mca_mark_as_used .....	287
mca_mark_as_unused .....	288
mca_get_adapter_name.....	289
mca_isadapter .....	290
mca_isenabled.....	291
10.4.2. MCA Bus DMA.....	292
mca_enable_dma.....	292
mca_disable_dma.....	293
mca_set_dma_addr .....	294
mca_get_dma_addr .....	295
mca_set_dma_count.....	296
mca_get_dma_residue.....	297
mca_set_dma_io .....	298
mca_set_dma_mode.....	299
<b>11. The Device File System .....</b>	<b>301</b>

devfs_put .....	301
devfs_register .....	301
devfs_unregister .....	303
devfs_mk_symlink .....	304
devfs_mk_dir .....	305
devfs_get_handle .....	307
devfs_get_flags .....	308
devfs_get_maj_min .....	309
devfs_get_handle_from_inode .....	311
devfs_generate_path .....	311
devfs_get_ops .....	313
devfs_set_file_size .....	314
devfs_get_info .....	315
devfs_set_info .....	316
devfs_get_parent .....	317
devfs_get_first_child .....	317
devfs_get_next_sibling .....	318
devfs_auto_unregister .....	319
devfs_get_unregister_slave .....	320
devfs_get_name .....	321
devfs_register_chrdev .....	322
devfs_register_blkdev .....	324
devfs_unregister_chrdev .....	325
devfs_unregister_blkdev .....	326
<b>12. Power Management .....</b>	<b>328</b>
pm_register .....	328
pm_unregister .....	329
pm_unregister_all .....	330
pm_send .....	331
pm_send_all .....	332
pm_find .....	334
<b>13. Block Devices .....</b>	<b>336</b>
blk_cleanup_queue .....	336

blk_queue_headactive .....	337
blk_queue_make_request.....	338
blk_init_queue.....	339
generic_make_request.....	341
submit_bh.....	343
ll_rw_block .....	344
end_that_request_first .....	345
<b>14. Miscellaneous Devices.....</b>	<b>348</b>
misc_register .....	348
misc_deregister .....	349
<b>15. Video4Linux .....</b>	<b>351</b>
video_unregister_device .....	351
<b>16. Sound Devices.....</b>	<b>352</b>
register_sound_special.....	352
register_sound_mixer.....	353
register_sound_midi.....	354
register_sound_dsp .....	355
register_sound_synth .....	356
unregister_sound_special.....	357
unregister_sound_mixer.....	358
unregister_sound_midi.....	359
unregister_sound_dsp .....	360
unregister_sound_synth .....	361
<b>17. USB Devices .....</b>	<b>363</b>
usb_register .....	363
usb_scan_devices .....	364
usb_deregister .....	365
usb_alloc_bus.....	365
usb_free_bus .....	366
usb_register_bus.....	367
usb_deregister_bus.....	368
usb_match_id .....	369

usb_alloc_urb.....	372
usb_free_urb .....	373
usb_control_msg .....	374
usb_bulk_msg .....	376
<b>18. 16x50 UART Driver .....</b>	<b>379</b>
register_serial.....	379
unregister_serial.....	380
<b>19. Z85230 Support Library .....</b>	<b>382</b>
z8530_interrupt.....	382
z8530_sync_open.....	383
z8530_sync_close .....	384
z8530_sync_dma_open.....	385
z8530_sync_dma_close .....	386
z8530_sync_txdma_open.....	387
z8530_sync_txdma_close .....	388
z8530_describe .....	389
z8530_init .....	390
z8530_shutdown .....	392
z8530_channel_load .....	393
z8530_null_rx .....	394
z8530_queue_xmit.....	394
z8530_get_stats.....	396
<b>20. Frame Buffer Library.....</b>	<b>398</b>
20.1. Frame Buffer Memory .....	398
register_framebuffer.....	398
unregister_framebuffer.....	399
20.2. Frame Buffer Console.....	400
fbcon_redraw_clear.....	400
fbcon_redraw_bmove.....	402
20.3. Frame Buffer Colormap .....	403
fb_alloc_cmap.....	404
fb_copy_cmap.....	405
fb_get_cmap.....	406

fb_set_cmap .....	407
fb_default_cmap .....	409
fb_invert_cmaps .....	410
20.4. Frame Buffer Generic Functions.....	411
fbgen_get_fix .....	411
fbgen_get_var .....	412
fbgen_set_var .....	413
fbgen_get_cmap.....	414
fbgen_set_cmap .....	416
fbgen_pan_display .....	417
fbgen_do_set_var .....	418
fbgen_set_disp .....	419
fbgen_install_cmap .....	420
fbgen_update_var.....	421
fbgen_switch .....	423
fbgen_blank.....	423
20.5. Frame Buffer Video Mode Database .....	424
fb_find_mode .....	425
__fb_try_mode.....	427
20.6. Frame Buffer Macintosh Video Mode Database.....	428
console_getmode.....	428
console_setmode .....	429
console_setcmap .....	430
console_powermode .....	431
mac_vmode_to_var .....	432
mac_var_to_vmode.....	434
mac_map_monitor_sense.....	435
mac_find_mode.....	436
20.7. Frame Buffer Fonts .....	437
fbcon_find_font.....	438
fbcon_get_default_font.....	438



# Chapter 1. Driver Basics

## 1.1. Driver Entry and Exit points

### **module\_init**

#### **Name**

`module_init` — driver initialization entry point

#### **Synopsis**

```
module_init ( x );
```

#### **Arguments**

`x`

function to be run at kernel boot time or module insertion

## Description

`module_init` will add the driver initialization routine in the “`__initcall.int`” code segment if the driver is checked as “y” or static, or else it will wrap the driver initialization routine with `init_module` which is used by `insmod` and `modprobe` when the driver is used as a module.

## `module_exit`

### Name

`module_exit` — driver exit entry point

### Synopsis

```
module_exit ( x );
```

### Arguments

`x`

function to be run when driver is removed

## Description

`module_exit` will wrap the driver clean-up code with `cleanup_module` when used with `rmmod` when the driver is a module. If the driver is statically compiled into the kernel, `module_exit` has no effect.

## 1.2. Atomic and pointer manipulation

### `atomic_read`

#### Name

`atomic_read` — read atomic variable

#### Synopsis

```
atomic_read ( v );
```

#### Arguments

`v`

pointer of type `atomic_t`

## Description

Atomically reads the value of  $v$ . Note that the guaranteed useful range of an `atomic_t` is only 24 bits.

## `atomic_set`

### Name

`atomic_set` — set atomic variable

### Synopsis

```
atomic_set ( v, i );
```

### Arguments

$v$   
pointer of type `atomic_t`

$i$   
required value

## Description

Atomically sets the value of  $v$  to  $i$ . Note that the guaranteed useful range of an `atomic_t` is only 24 bits.

## `atomic_add`

### Name

`atomic_add` — add integer to atomic variable

### Synopsis

```
void atomic_add (int  $i$ , atomic_t *  $v$ );
```

### Arguments

$i$	integer value to add
$v$	pointer of type <code>atomic_t</code>

## Description

Atomically adds *i* to *v*. Note that the guaranteed useful range of an `atomic_t` is only 24 bits.

## `atomic_sub`

### Name

`atomic_sub` — subtract the atomic variable

### Synopsis

```
void atomic_sub (int i, atomic_t * v);
```

### Arguments

*i*  
integer value to subtract

*v*  
pointer of type `atomic_t`

## Description

Atomically subtracts  $i$  from  $v$ . Note that the guaranteed useful range of an `atomic_t` is only 24 bits.

# atomic\_sub\_and\_test

## Name

`atomic_sub_and_test` — subtract value from variable and test result

## Synopsis

```
int atomic_sub_and_test (int  $i$ , atomic_t *  $v$ );
```

## Arguments

$i$   
integer value to subtract

$v$   
pointer of type `atomic_t`

## Description

Atomically subtracts  $i$  from  $v$  and returns true if the result is zero, or false for all other cases. Note that the guaranteed useful range of an `atomic_t` is only 24 bits.

## `atomic_inc`

### Name

`atomic_inc` — increment atomic variable

### Synopsis

```
void atomic_inc (atomic_t * v);
```

### Arguments

$v$   
pointer of type `atomic_t`



## Description

Atomically increments  $v$  by 1. Note that the guaranteed useful range of an `atomic_t` is only 24 bits.

## `atomic_dec`

### Name

`atomic_dec` — decrement atomic variable

### Synopsis

```
void atomic_dec (atomic_t * v);
```

### Arguments

$v$   
pointer of type `atomic_t`

## Description

Atomically decrements  $v$  by 1. Note that the guaranteed useful range of an `atomic_t` is only 24 bits.

## `atomic_dec_and_test`

### Name

`atomic_dec_and_test` — decrement and test

### Synopsis

```
int atomic_dec_and_test (atomic_t * v);
```

### Arguments

$v$

pointer of type `atomic_t`

## Description

Atomically decrements  $v$  by 1 and returns true if the result is 0, or false for all other cases. Note that the guaranteed useful range of an `atomic_t` is only 24 bits.

# atomic\_inc\_and\_test

## Name

`atomic_inc_and_test` — increment and test

## Synopsis

```
int atomic_inc_and_test (atomic_t * v);
```

## Arguments

$v$

pointer of type `atomic_t`

## Description

Atomically increments  $v$  by 1 and returns true if the result is zero, or false for all other cases. Note that the guaranteed useful range of an `atomic_t` is only 24 bits.

# atomic\_add\_negative

## Name

`atomic_add_negative` — add and test if negative

## Synopsis

```
int atomic_add_negative (int i, atomic_t * v);
```

## Arguments

*i*

integer value to add

*v*

pointer of type `atomic_t`

## Description

Atomically adds  $i$  to  $v$  and returns true if the result is negative, or false when result is greater than or equal to zero. Note that the guaranteed useful range of an `atomic_t` is only 24 bits.

## get\_unaligned

### Name

`get_unaligned` — get value from possibly mis-aligned location

### Synopsis

```
get_unaligned ( ptr );
```

### Arguments

*ptr*

pointer to value

## Description

This macro should be used for accessing values larger in size than single bytes at locations that are expected to be improperly aligned, e.g. retrieving a u16 value from a location not u16-aligned.

Note that unaligned accesses can be very expensive on some architectures.

## put\_unaligned

### Name

`put_unaligned` — put value to a possibly mis-aligned location

### Synopsis

```
put_unaligned ( val, ptr);
```

### Arguments

*val*

value to place

*ptr*

pointer to location

## Description

This macro should be used for placing values larger in size than single bytes at locations that are expected to be improperly aligned, e.g. writing a u16 value to a location not u16-aligned.

Note that unaligned accesses can be very expensive on some architectures.

## 1.3. Delaying, scheduling, and timer routines

### **schedule\_timeout**

#### **Name**

`schedule_timeout` — sleep until timeout

#### **Synopsis**

```
signed long schedule_timeout (signed long timeout);
```

## Arguments

*timeout*

timeout value in jiffies

## Description

Make the current task sleep until *timeout* jiffies have elapsed. The routine will return immediately unless the current task state has been set (see `set_current_state`).

You can set the task state as follows -

`TASK_UNINTERRUPTIBLE` - at least *timeout* jiffies are guaranteed to pass before the routine returns. The routine will return 0

`TASK_INTERRUPTIBLE` - the routine may return early if a signal is delivered to the current task. In this case the remaining time in jiffies will be returned, or 0 if the timer expired in time

The current task state is guaranteed to be `TASK_RUNNING` when this routine returns.

Specifying a *timeout* value of `MAX_SCHEDULE_TIMEOUT` will schedule the CPU away without a bound on the timeout. In this case the return value will be `MAX_SCHEDULE_TIMEOUT`.

In all cases the return value is guaranteed to be non-negative.



# reparent\_to\_init

## Name

`reparent_to_init` — Reparent the calling kernel thread to the init task.

## Synopsis

```
void reparent_to_init ( void );
```

## Arguments

*void*

no arguments

## Description

If a kernel thread is launched as a result of a system call, or if it ever exits, it should generally reparent itself to init so that it is correctly cleaned up on exit.

The various task state such as scheduling policy and priority may have been inherited from a user process, so we reset them to sane values here.

NOTE that `reparent_to_init` gives the caller full capabilities.

# Chapter 2. Data Types

## 2.1. Doubly Linked Lists

### **list\_add**

#### **Name**

`list_add` — add a new entry

#### **Synopsis**

```
void list_add (struct list_head * new, struct list_head * head);
```

#### **Arguments**

*new*

new entry to be added

*head*

list head to add it after

## Description

Insert a new entry after the specified head. This is good for implementing stacks.

# list\_add\_tail

## Name

`list_add_tail` — add a new entry

## Synopsis

```
void list_add_tail (struct list_head * new, struct list_head *  
head);
```

## Arguments

*new*

new entry to be added

*head*

list head to add it before

## Description

Insert a new entry before the specified head. This is useful for implementing queues.

# list\_del

## Name

`list_del` — deletes entry from list.

## Synopsis

```
void list_del (struct list_head * entry);
```

## Arguments

*entry*

the element to delete from the list.

## Note

`list_empty` on `entry` does not return true after this, the entry is in an undefined state.

## **list\_del\_init**

### **Name**

`list_del_init` — deletes entry from list and reinitialize it.

### **Synopsis**

```
void list_del_init (struct list_head * entry);
```

### **Arguments**

*entry*

the element to delete from the list.

## **list\_empty**

### **Name**

`list_empty` — tests whether a list is empty

### **Synopsis**

```
int list_empty (struct list_head * head);
```

### **Arguments**

*head*

the list to test.

## **list\_splice**

### **Name**

`list_splice` — join two lists

## Synopsis

```
void list_splice (struct list_head * list, struct list_head *  
head);
```

## Arguments

*list*

the new list to add.

*head*

the place to add it in the first list.

## list\_entry

### Name

`list_entry` — get the struct for this entry

## Synopsis

```
list_entry ( ptr, type, member );
```

## Arguments

*ptr*

the &struct list\_head pointer.

*type*

the type of the struct this is embedded in.

*member*

the name of the list\_struct within the struct.

## list\_for\_each

### Name

list\_for\_each — iterate over a list

### Synopsis

```
list_for_each ( pos, head );
```



## Arguments

*pos*

the &struct list\_head to use as a loop counter.

*head*

the head for your list.

## list\_for\_each\_safe

### Name

`list_for_each_safe` — iterate over a list safe against removal of list entry

### Synopsis

```
list_for_each_safe ( pos,  n,  head );
```

## Arguments

*pos*

the &struct list\_head to use as a loop counter.

*n*

another `&struct list_head` to use as temporary storage

*head*

the head for your list.

## **list\_for\_each\_prev**

### **Name**

`list_for_each_prev` — iterate over a list in reverse order

### **Synopsis**

```
list_for_each_prev ( pos, head );
```

### **Arguments**

*pos*

the `&struct list_head` to use as a loop counter.

*head*

the head for your list.



# Chapter 3. Basic C Library Functions

When writing drivers, you cannot in general use routines which are from the C Library. Some of the functions have been found generally useful and they are listed below. The behaviour of these functions may vary slightly from those defined by ANSI, and these deviations are noted in the text.

## 3.1. String Conversions

### simple\_strtoll

#### Name

`simple_strtoll` — convert a string to a signed long long

#### Synopsis

```
long long simple_strtoll (const char * cp, char ** endp,  
unsigned int base);
```

#### Arguments

*cp*

The start of the string

*endp*

A pointer to the end of the parsed string will be placed here

*base*

The number base to use

## simple\_strtoul

### Name

`simple_strtoul` — convert a string to an unsigned long

### Synopsis

```
unsigned long simple_strtoul (const char * cp, char ** endp,  
unsigned int base);
```

### Arguments

*cp*

The start of the string

*endp*

A pointer to the end of the parsed string will be placed here

*base*

The number base to use

## simple\_strtol

### Name

`simple_strtol` — convert a string to a signed long

### Synopsis

```
long simple_strtol (const char * cp, char ** endp, unsigned int  
base);
```

### Arguments

*cp*

The start of the string

*endp*

A pointer to the end of the parsed string will be placed here

*base*

The number base to use

## simple\_strtoul

### Name

simple\_strtoul — convert a string to an unsigned long long

### Synopsis

```
unsigned long long simple_strtoul (const char * cp, char **  
endp, unsigned int base);
```

### Arguments

*cp*

The start of the string

*endp*

A pointer to the end of the parsed string will be placed here

*base*

The number base to use

## **vsnprintf**

### **Name**

`vsnprintf` — Format a string and place it in a buffer

### **Synopsis**

```
int vsnprintf (char * buf, size_t size, const char * fmt,  
va_list args);
```

### **Arguments**

*buf*

The buffer to place the result into



*size*

The size of the buffer, including the trailing null space

*fmt*

The format string to use

*args*

Arguments for the format string

## Description

Call this function if you are already dealing with a `va_list`. You probably want `snprintf` instead.

# snprintf

## Name

`snprintf` — Format a string and place it in a buffer

## Synopsis

```
int snprintf (char * buf, size_t size, const char * fmt, ...  
...);
```

## Arguments

*buf*

The buffer to place the result into

*size*

The size of the buffer, including the trailing null space

*fmt*

The format string to use @...: Arguments for the format string

...

variable arguments

## vsprintf

### Name

`vsprintf` — Format a string and place it in a buffer

### Synopsis

```
int vsprintf (char * buf, const char * fmt, va_list args);
```

## Arguments

*buf*

The buffer to place the result into

*fmt*

The format string to use

*args*

Arguments for the format string

## Description

Call this function if you are already dealing with a `va_list`. You probably want `sprintf` instead.

# sprintf

## Name

`sprintf` — Format a string and place it in a buffer

## Synopsis

```
int sprintf (char * buf, const char * fmt, ... ...);
```

## Arguments

*buf*

The buffer to place the result into

*fmt*

The format string to use @...: Arguments for the format string

...

variable arguments

## vsscanf

### Name

`vsscanf` — Unformat a buffer into a list of arguments

### Synopsis

```
int vsscanf (const char * buf, const char * fmt, va_list args);
```

## Arguments

*buf*

input buffer

*fmt*

format of buffer

*args*

arguments

## sscanf

### Name

`sscanf` — Unformat a buffer into a list of arguments

### Synopsis

```
int sscanf (const char * buf, const char * fmt, ... ..);
```

## Arguments

*buf*

input buffer

*fmt*

formatting of buffer @...: resulting arguments

...

variable arguments

## 3.2. String Manipulation

### strcpy

#### Name

strcpy — Copy a NUL terminated string

#### Synopsis

```
char * strcpy (char * dest, const char * src);
```

## Arguments

*dest*

Where to copy the string to

*src*

Where to copy the string from

## strncpy

### Name

`strncpy` — Copy a length-limited, NUL-terminated string

### Synopsis

```
char * strncpy (char * dest, const char * src, size_t count);
```

## Arguments

*dest*

Where to copy the string to

*src*

Where to copy the string from

*count*

The maximum number of bytes to copy

## Description

Note that unlike userspace `strncpy`, this does not NUL-pad the buffer. However, the result is not NUL-terminated if the source exceeds *count* bytes.

# strcat

## Name

`strcat` — Append one NUL-terminated string to another

## Synopsis

```
char * strcat (char * dest, const char * src);
```



## Arguments

*dest*

The string to be appended to

*src*

The string to append to it

## strncat

### Name

`strncat` — Append a length-limited, NUL-terminated string to another

### Synopsis

```
char * strncat (char * dest, const char * src, size_t count);
```

## Arguments

*dest*

The string to be appended to

*src*

The string to append to it

*count*

The maximum numbers of bytes to copy

## Description

Note that in contrast to `strncpy`, `strncat` ensures the result is terminated.

## strcmp

### Name

`strcmp` — Compare two strings

### Synopsis

```
int strcmp (const char * cs, const char * ct);
```

## Arguments

*cs*

One string

*ct*

Another string

## strncmp

### Name

`strncmp` — Compare two length-limited strings

### Synopsis

```
int strncmp (const char * cs, const char * ct, size_t count);
```

## Arguments

*cs*

One string

*ct*

Another string

*count*

The maximum number of bytes to compare

## strchr

### Name

`strchr` — Find the first occurrence of a character in a string

### Synopsis

```
char * strchr (const char * s, int c);
```

### Arguments

*s*

The string to be searched

*c*

The character to search for

# strrchr

## Name

`strrchr` — Find the last occurrence of a character in a string

## Synopsis

```
char * strrchr (const char * s, int c);
```

## Arguments

*s*

The string to be searched

*c*

The character to search for

## strlen

### Name

`strlen` — Find the length of a string

### Synopsis

```
size_t strlen (const char * s);
```

### Arguments

*s*

The string to be sized

## strnlen

### Name

`strnlen` — Find the length of a length-limited string

## Synopsis

```
size_t strnlen (const char * s, size_t count);
```

## Arguments

*s*

The string to be sized

*count*

The maximum number of bytes to search

## strpbrk

### Name

`strpbrk` — Find the first occurrence of a set of characters

## Synopsis

```
char * strpbrk (const char * cs, const char * ct);
```

## Arguments

*cs*

The string to be searched

*ct*

The characters to search for

## strtok

### Name

`strtok` — Split a string into tokens

### Synopsis

```
char * strtok (char * s, const char * ct);
```

## Arguments

*s*

The string to be searched



*ct*

The characters to search for

## WARNING

strtok is deprecated, use strsep instead.

# memset

## Name

memset — Fill a region of memory with the given value

## Synopsis

```
void * memset (void * s, int c, size_t count);
```

## Arguments

*s*

Pointer to the start of the area.

*c*

The byte to fill the area with

*count*

The size of the area.

## Description

Do not use `memset` to access IO space, use `memset_io` instead.

# bcopy

## Name

`bcopy` — Copy one area of memory to another

## Synopsis

```
char * bcopy (const char * src, char * dest, int count);
```

## Arguments

*src*

Where to copy from

*dest*

Where to copy to

*count*

The size of the area.

## Description

Note that this is the same as `memcpy`, with the arguments reversed. `memcpy` is the standard, `bcopy` is a legacy BSD function.

You should not use this function to access IO space, use `memcpy_toio` or `memcpy_fromio` instead.

## `memcpy`

### Name

`memcpy` — Copy one area of memory to another

## Synopsis

```
void * memcpy (void * dest, const void * src, size_t count);
```

## Arguments

*dest*

Where to copy to

*src*

Where to copy from

*count*

The size of the area.

## Description

You should not use this function to access IO space, use `memcpy_toio` or `memcpy_fromio` instead.

# memmove

## Name

memmove — Copy one area of memory to another

## Synopsis

```
void * memmove (void * dest, const void * src, size_t count);
```

## Arguments

*dest*

Where to copy to

*src*

Where to copy from

*count*

The size of the area.

## Description

Unlike `memcpy`, `memmove` copes with overlapping areas.

# memcmp

## Name

memcmp — Compare two areas of memory

## Synopsis

```
int memcmp (const void * cs, const void * ct, size_t count);
```

## Arguments

*cs*

One area of memory

*ct*

Another area of memory

*count*

The size of the area.

# memscan

## Name

`memscan` — Find a character in an area of memory.

## Synopsis

```
void * memscan (void * addr, int c, size_t size);
```

## Arguments

*addr*

The memory area

*c*

The byte to search for

*size*

The size of the area.

## Description

returns the address of the first occurrence of *c*, or 1 byte past the area if *c* is not found

# strstr

## Name

`strstr` — Find the first substring in a NUL terminated string

## Synopsis

```
char * strstr (const char * s1, const char * s2);
```

## Arguments

*s1*

The string to be searched

*s2*

The string to search for



# memchr

## Name

`memchr` — Find a character in an area of memory.

## Synopsis

```
void * memchr (const void * s, int c, size_t n);
```

## Arguments

*s*

The memory area

*c*

The byte to search for

*n*

The size of the area.

## Description

returns the address of the first occurrence of *c*, or `NULL` if *c* is not found

## 3.3. Bit Operations

### set\_bit

#### Name

`set_bit` — Atomically set a bit in memory

#### Synopsis

```
void set_bit (int nr, volatile void * addr);
```

#### Arguments

*nr*

the bit to set

*addr*

the address to start counting from

## Description

This function is atomic and may not be reordered. See `__set_bit` if you do not require the atomic guarantees. Note that *nr* may be almost arbitrarily large; this function is not restricted to acting on a single-word quantity.

## `__set_bit`

### Name

`__set_bit` — Set a bit in memory

### Synopsis

```
void __set_bit (int nr, volatile void * addr);
```

### Arguments

*nr*

the bit to set

*addr*

the address to start counting from

## Description

Unlike `set_bit`, this function is non-atomic and may be reordered. If it's called on the same region of memory simultaneously, the effect may be that only one operation succeeds.

## clear\_bit

### Name

`clear_bit` — Clears a bit in memory

### Synopsis

```
void clear_bit (int nr, volatile void * addr);
```

### Arguments

*nr*

Bit to clear

*addr*

Address to start counting from

## Description

`clear_bit` is atomic and may not be reordered. However, it does not contain a memory barrier, so if it is used for locking purposes, you should call `smp_mb__before_clear_bit` and/or `smp_mb__after_clear_bit` in order to ensure changes are visible on other processors.

## `__change_bit`

### Name

`__change_bit` — Toggle a bit in memory

### Synopsis

```
void __change_bit (int nr, volatile void * addr);
```

### Arguments

*nr*

the bit to set

*addr*

the address to start counting from

## Description

Unlike `change_bit`, this function is non-atomic and may be reordered. If it's called on the same region of memory simultaneously, the effect may be that only one operation succeeds.

# change\_bit

## Name

`change_bit` — Toggle a bit in memory

## Synopsis

```
void change_bit (int nr, volatile void * addr);
```

## Arguments

*nr*

Bit to clear

*addr*

Address to start counting from

## Description

`change_bit` is atomic and may not be reordered. Note that *nr* may be almost arbitrarily large; this function is not restricted to acting on a single-word quantity.

## test\_and\_set\_bit

### Name

`test_and_set_bit` — Set a bit and return its old value

### Synopsis

```
int test_and_set_bit (int nr, volatile void * addr);
```

### Arguments

*nr*

Bit to set

*addr*

Address to count from

## Description

This operation is atomic and cannot be reordered. It also implies a memory barrier.

# **\_\_test\_and\_set\_bit**

## Name

`__test_and_set_bit` — Set a bit and return its old value

## Synopsis

```
int __test_and_set_bit (int nr, volatile void * addr);
```

## Arguments

*nr*

Bit to set

*addr*

Address to count from



## Description

This operation is non-atomic and can be reordered. If two examples of this operation race, one can appear to succeed but actually fail. You must protect multiple accesses with a lock.

## test\_and\_clear\_bit

### Name

`test_and_clear_bit` — Clear a bit and return its old value

### Synopsis

```
int test_and_clear_bit (int nr, volatile void * addr);
```

### Arguments

*nr*

Bit to set

*addr*

Address to count from

## Description

This operation is atomic and cannot be reordered. It also implies a memory barrier.

# `__test_and_clear_bit`

## Name

`__test_and_clear_bit` — Clear a bit and return its old value

## Synopsis

```
int __test_and_clear_bit (int nr, volatile void * addr);
```

## Arguments

*nr*

Bit to set

*addr*

Address to count from

## Description

This operation is non-atomic and can be reordered. If two examples of this operation race, one can appear to succeed but actually fail. You must protect multiple accesses with a lock.

# test\_and\_change\_bit

## Name

`test_and_change_bit` — Change a bit and return its new value

## Synopsis

```
int test_and_change_bit (int nr, volatile void * addr);
```

## Arguments

*nr*

Bit to set

*addr*

Address to count from

## Description

This operation is atomic and cannot be reordered. It also implies a memory barrier.

## test\_bit

### Name

`test_bit` — Determine whether a bit is set

### Synopsis

```
int test_bit (int nr, const volatile void * addr);
```

### Arguments

*nr*

bit number to test

*addr*

Address to start counting from

# find\_first\_zero\_bit

## Name

`find_first_zero_bit` — find the first zero bit in a memory region

## Synopsis

```
int find_first_zero_bit (void * addr, unsigned size);
```

## Arguments

*addr*

The address to start the search at

*size*

The maximum size to search

## Description

Returns the bit-number of the first zero bit, not the number of the byte containing a bit.

# find\_next\_zero\_bit

## Name

`find_next_zero_bit` — find the first zero bit in a memory region

## Synopsis

```
int find_next_zero_bit (void * addr, int size, int offset);
```

## Arguments

*addr*

The address to base the search on

*size*

The maximum size to search

*offset*

The bitnumber to start searching at

# ffz

## Name

`ffz` — find first zero in word.

## Synopsis

```
unsigned long ffz (unsigned long word);
```

## Arguments

*word*

The word to search

## Description

Undefined if no zero exists, so code should check against `~0UL` first.

# ffs

## Name

`ffs` — find first bit set

## Synopsis

```
int ffs (int x);
```

## Arguments

`x`

the word to search

## Description

This is defined the same way as the libc and compiler builtin ffs routines, therefore differs in spirit from the above ffz (man ffs).



# **hweight32**

## **Name**

`hweight32` — returns the hamming weight of a N-bit word

## **Synopsis**

```
hweight32 ( x );
```

## **Arguments**

`x`

the word to weigh

## **Description**

The Hamming Weight of a number is the total number of bits set in it.

# Chapter 4. Memory Management in Linux

## 4.1. The Slab Cache

### **kmem\_cache\_create**

#### **Name**

`kmem_cache_create` — Create a cache.

#### **Synopsis**

```
kmem_cache_t * kmem_cache_create (const char * name, size_t
size, size_t offset, unsigned long flags, void (*ctor) (void*,
kmem_cache_t *, unsigned long), void (*dtor) (void*,
kmem_cache_t *, unsigned long));
```

#### **Arguments**

*name*

A string which is used in `/proc/slabinfo` to identify this cache.

*size*

The size of objects to be created in this cache.

*offset*

The offset to use within the page.

*flags*

SLAB flags

*ctor*

A constructor for the objects.

*dtor*

A destructor for the objects.

## Description

Returns a ptr to the cache on success, NULL on failure. Cannot be called within a int, but can be interrupted. The *ctor* is run when new pages are allocated by the cache and the *dtor* is run before the pages are handed back. The flags are

SLAB\_POISON - Poison the slab with a known test pattern (a5a5a5a5) to catch references to uninitialised memory.

SLAB\_RED\_ZONE - Insert 'Red' zones around the allocated memory to check for buffer overruns.

SLAB\_NO\_REAP - Don't automatically reap this cache when we're under memory pressure.

SLAB\_HWCACHE\_ALIGN - Align the objects in this cache to a hardware cacheline. This can be beneficial if you're counting cycles as closely as davem.

# **kmem\_cache\_shrink**

## **Name**

`kmem_cache_shrink` — Shrink a cache.

## **Synopsis**

```
int kmem_cache_shrink (kmem_cache_t * cachep);
```

## **Arguments**

*cachep*

The cache to shrink.

## **Description**

Releases as many slabs as possible for a cache. To help debugging, a zero exit status indicates all slabs were released.

# kmem\_cache\_destroy

## Name

`kmem_cache_destroy` — delete a cache

## Synopsis

```
int kmem_cache_destroy (kmem_cache_t * cachep);
```

## Arguments

*cachep*

the cache to destroy

## Description

Remove a `kmem_cache_t` object from the slab cache. Returns 0 on success.

It is expected this function will be called by a module when it is unloaded. This will remove the cache completely, and avoid a duplicate cache being allocated each time a module is loaded and unloaded, if the module doesn't have persistent in-kernel storage across loads and unloads.

The caller must guarantee that noone will allocate memory from the cache during the `kmem_cache_destroy`.

# kmem\_cache\_alloc

## Name

`kmem_cache_alloc` — Allocate an object

## Synopsis

```
void * kmem_cache_alloc (kmem_cache_t * cachep, int flags);
```

## Arguments

*cachep*

The cache to allocate from.

*flags*

See `kmalloc`.

## Description

Allocate an object from this cache. The flags are only relevant if the cache has no available objects.

# kmalloc

## Name

`kmalloc` — allocate memory

## Synopsis

```
void * kmalloc (size_t size, int flags);
```

## Arguments

*size*

how many bytes of memory are required.

*flags*

the type of memory to allocate.

## Description

`kmalloc` is the normal method of allocating memory in the kernel.

The *flags* argument may be one of:

GFP\_USER - Allocate memory on behalf of user. May sleep.

GFP\_KERNEL - Allocate normal kernel ram. May sleep.

GFP\_ATOMIC - Allocation will not sleep. Use inside interrupt handlers.

Additionally, the GFP\_DMA flag may be set to indicate the memory must be suitable for DMA. This can mean different things on different platforms. For example, on i386, it means that the memory must come from the first 16MB.

## kmem\_cache\_free

### Name

`kmem_cache_free` — Deallocate an object

### Synopsis

```
void kmem_cache_free (kmem_cache_t * cachep, void * objp);
```

### Arguments

*cachep*

The cache the allocation was from.



*objp*

The previously allocated object.

## Description

Free an object which was previously allocated from this cache.

# kfree

## Name

kfree — free previously allocated memory

## Synopsis

```
void kfree (const void * objp);
```

## Arguments

*objp*

pointer returned by kmalloc.

## **Description**

Don't free memory not originally allocated by `kmalloc` or you will run into trouble.

# Chapter 5. The proc filesystem

## 5.1. sysctl interface

### register\_sysctl\_table

#### Name

`register_sysctl_table` — register a sysctl heirarchy

#### Synopsis

```
struct ctl_table_header * register_sysctl_table (ctl_table *  
table, int insert_at_head);
```

#### Arguments

*table*

the top-level table structure

*insert\_at\_head*

whether the entry should be inserted in front or at the end

## Description

Register a sysctl table heirarchy. *table* should be a filled in `ctl_table` array. An entry with a `ctl_name` of 0 terminates the table.

The members of the `&ctl_table` structure are used as follows:

`ctl_name` - This is the numeric sysctl value used by `sysctl(2)`. The number must be unique within that level of sysctl

`procname` - the name of the sysctl file under `/proc/sys`. Set to `NULL` to not enter a sysctl file

`data` - a pointer to data for use by `proc_handler`

`maxlen` - the maximum size in bytes of the data

`mode` - the file permissions for the `/proc/sys` file, and for `sysctl(2)`

`child` - a pointer to the child sysctl table if this entry is a directory, or `NULL`.

`proc_handler` - the text handler routine (described below)

`strategy` - the strategy routine (described below)

`de` - for internal use by the sysctl routines

`extra1`, `extra2` - extra pointers usable by the proc handler routines

Leaf nodes in the sysctl tree will be represented by a single file under `/proc`; non-leaf nodes will be represented by directories.

`sysctl(2)` can automatically manage read and write requests through the sysctl table.

The `data` and `maxlen` fields of the `ctl_table` struct enable minimal validation of the values being written to be performed, and the `mode` field allows minimal authentication.

More sophisticated management can be enabled by the provision of a strategy routine with the table entry. This will be called before any automatic read or write of the data is performed.

The strategy routine may return

< 0 - Error occurred (error is passed to user process)

0 - OK - proceed with automatic read or write.

> 0 - OK - read or write has been done by the strategy routine, so return immediately.

There must be a `proc_handler` routine for any terminal nodes mirrored under `/proc/sys` (non-terminals are handled by a built-in directory handler). Several default handlers are available to cover common cases -

```
proc_doststring, proc_dointvec, proc_dointvec_jiffies,  
proc_dointvec_minmax, proc_doulongvec_ms_jiffies_minmax,  
proc_doulongvec_minmax
```

It is the handler's job to read the input buffer from user memory and process it. The handler should return 0 on success.

This routine returns `NULL` on a failure to register, and a pointer to the table header on success.

## unregister\_sysctl\_table

### Name

`unregister_sysctl_table` — unregister a sysctl table heirarchy

### Synopsis

```
void unregister_sysctl_table (struct ctl_table_header * header);
```

## Arguments

*header*

the header returned from `register_sysctl_table`

## Description

Unregisters the sysctl table and all children. `proc` entries may not actually be removed until they are no longer used by anyone.

# proc\_dostring

## Name

`proc_dostring` — read a string sysctl

## Synopsis

```
int proc_dostring (ctl_table * table, int write, struct file *  
filp, void * buffer, size_t * lenp);
```

## Arguments

*table*

the sysctl table

*write*

TRUE if this is a write to the sysctl file

*filp*

the file structure

*buffer*

the user buffer

*lenp*

the size of the user buffer

## Description

Reads/writes a string from/to the user buffer. If the kernel buffer provided is not large enough to hold the string, the string is truncated. The copied string is NULL-terminated. If the string is being read by the user process, it is copied and a newline '\n' is added. It is truncated if the buffer is not large enough.

Returns 0 on success.

# proc\_dointvec

## Name

`proc_dointvec` — read a vector of integers

## Synopsis

```
int proc_dointvec (ctl_table * table, int write, struct file *  
filp, void * buffer, size_t * lenp);
```

## Arguments

*table*

the sysctl table

*write*

TRUE if this is a write to the sysctl file

*filp*

the file structure

*buffer*

the user buffer



*lenp*

the size of the user buffer

## Description

Reads/writes up to `table->maxlen/sizeof(unsigned int)` integer values from/to the user buffer, treated as an ASCII string.

Returns 0 on success.

# proc\_dointvec\_minmax

## Name

`proc_dointvec_minmax` — read a vector of integers with min/max values

## Synopsis

```
int proc_dointvec_minmax (ctl_table * table, int write, struct  
file * filp, void * buffer, size_t * lenp);
```

## Arguments

*table*

the sysctl table

*write*

TRUE if this is a write to the sysctl file

*filp*

the file structure

*buffer*

the user buffer

*lenp*

the size of the user buffer

## Description

Reads/writes up to `table->maxlen/sizeof(unsigned int)` integer values from/to the user buffer, treated as an ASCII string.

This routine will ensure the values are within the range specified by `table->extra1` (min) and `table->extra2` (max).

Returns 0 on success.

## proc\_doulongvec\_minmax

### Name

`proc_doulongvec_minmax` — read a vector of long integers with min/max values

### Synopsis

```
int proc_doulongvec_minmax (ctl_table * table, int write, struct  
file * filp, void * buffer, size_t * lenp);
```

### Arguments

*table*

the sysctl table

*write*

TRUE if this is a write to the sysctl file

*filp*

the file structure

*buffer*

the user buffer

*lenp*

the size of the user buffer

## Description

Reads/writes up to `table->maxlen/sizeof(unsigned long)` unsigned long values from/to the user buffer, treated as an ASCII string.

This routine will ensure the values are within the range specified by `table->extra1` (min) and `table->extra2` (max).

Returns 0 on success.

# proc\_doulongvec\_ms\_jiffies\_minmax

## Name

`proc_doulongvec_ms_jiffies_minmax` — read a vector of millisecond values with min/max values

## Synopsis

```
int proc_doulongvec_ms_jiffies_minmax (ctl_table * table, int  
write, struct file * filp, void * buffer, size_t * lenp);
```

## Arguments

*table*

the sysctl table

*write*

TRUE if this is a write to the sysctl file

*filp*

the file structure

*buffer*

the user buffer

*lenp*

the size of the user buffer

## Description

Reads/writes up to `table->maxlen/sizeof(unsigned long)` unsigned long values from/to the user buffer, treated as an ASCII string. The values are treated as milliseconds, and converted to jiffies when they are stored.

This routine will ensure the values are within the range specified by `table->extra1` (min) and `table->extra2` (max).

Returns 0 on success.

## proc\_dointvec\_jiffies

### Name

`proc_dointvec_jiffies` — read a vector of integers as seconds

### Synopsis

```
int proc_dointvec_jiffies (ctl_table * table, int write, struct  
file * filp, void * buffer, size_t * lenp);
```

### Arguments

*table*

the sysctl table

*write*

TRUE if this is a write to the sysctl file

*filp*

the file structure

*buffer*

the user buffer

*lenp*

the size of the user buffer

## **Description**

Reads/writes up to `table->maxlen/sizeof(unsigned int)` integer values from/to the user buffer, treated as an ASCII string. The values read are assumed to be in seconds, and are converted into jiffies.

Returns 0 on success.

# Chapter 6. The Linux VFS

## 6.1. The Directory Cache

### **d\_invalidate**

#### **Name**

`d_invalidate` — invalidate a dentry

#### **Synopsis**

```
int d_invalidate (struct dentry * dentry);
```

#### **Arguments**

*dentry*

dentry to invalidate



## Description

Try to invalidate the dentry if it turns out to be possible. If there are other dentries that can be reached through this one we can't delete it and we return -EBUSY. On success we return 0.

no dcache lock.

## d\_find\_alias

### Name

`d_find_alias` — grab a hashed alias of inode

### Synopsis

```
struct dentry * d_find_alias (struct inode * inode);
```

### Arguments

*inode*

inode in question

## Description

If inode has a hashed alias - acquire the reference to alias and return it. Otherwise return NULL. Notice that if inode is a directory there can be only one alias and it can be unhashed only if it has no children.

# prune\_dcache

## Name

`prune_dcache` — shrink the dcache

## Synopsis

```
void prune_dcache (int count);
```

## Arguments

*count*

number of entries to try and free

## Description

Shrink the dcache. This is done when we need more memory, or simply when we need to unmount something (at which point we need to unuse all dentries).

This function may fail to free any resources if all the dentries are in use.

# shrink\_dcache\_sb

## Name

`shrink_dcache_sb` — shrink dcache for a superblock

## Synopsis

```
void shrink_dcache_sb (struct super_block * sb);
```

## Arguments

*sb*

superblock

## Description

Shrink the dcache for the specified super block. This is used to free the dcache before unmounting a file system

# have\_submounts

## Name

`have_submounts` — check for mounts over a dentry

## Synopsis

```
int have_submounts (struct dentry * parent);
```

## Arguments

*parent*

dentry to check.

## Description

Return true if the parent or its subdirectories contain a mount point

# shrink\_dcache\_parent

## Name

`shrink_dcache_parent` — prune dcache

## Synopsis

```
void shrink_dcache_parent (struct dentry * parent);
```

## Arguments

*parent*

parent of entries to prune

## Description

Prune the dcache to remove unused children of the parent dentry.

## d\_alloc

### Name

`d_alloc` — allocate a dcache entry

### Synopsis

```
struct dentry * d_alloc (struct dentry * parent, const struct  
qstr * name);
```

### Arguments

*parent*

parent of entry to allocate

*name*

qstr of the name

### Description

Allocates a dentry. It returns `NULL` if there is insufficient memory available. On a success the dentry is returned. The name passed in is copied and the copy passed in may be reused after this call.

# d\_instantiate

## Name

`d_instantiate` — fill in inode information for a dentry

## Synopsis

```
void d_instantiate (struct dentry * entry, struct inode *  
inode);
```

## Arguments

*entry*

dentry to complete

*inode*

inode to attach to this dentry

## Description

Fill in inode information in the entry.

This turns negative dentries into productive full members of society.

NOTE! This assumes that the inode count has been incremented (or otherwise set) by the caller to indicate that it is now in use by the dcache.

## d\_alloc\_root

### Name

`d_alloc_root` — allocate root dentry

### Synopsis

```
struct dentry * d_alloc_root (struct inode * root_inode);
```

### Arguments

*root\_inode*

inode to allocate the root for

### Description

Allocate a root (“/”) dentry for the inode given. The inode is instantiated and returned. `NULL` is returned if there is insufficient memory or the inode passed is `NULL`.



# d\_lookup

## Name

`d_lookup` — search for a dentry

## Synopsis

```
struct dentry * d_lookup (struct dentry * parent, struct qstr *  
name);
```

## Arguments

*parent*

parent dentry

*name*

qstr of name we wish to find

## Description

Searches the children of the parent dentry for the name in question. If the dentry is found its reference count is incremented and the dentry is returned. The caller must use `d_put` to free the entry when it has finished using it. `NULL` is returned on failure.

# d\_validate

## Name

`d_validate` — verify dentry provided from insecure source

## Synopsis

```
int d_validate (struct dentry * dentry, struct dentry *  
dparent);
```

## Arguments

*dentry*

The dentry alleged to be valid child of *dparent*

*dparent*

The parent dentry (known to be valid)

## Description

An insecure source has sent us a dentry, here we verify it and `dget` it. This is used by `ncpfs` in its `readdir` implementation. Zero is returned in the dentry is invalid.

# d\_delete

## Name

`d_delete` — delete a dentry

## Synopsis

```
void d_delete (struct dentry * dentry);
```

## Arguments

*dentry*

The dentry to delete

## Description

Turn the dentry into a negative dentry if possible, otherwise remove it from the hash queues so it can be deleted later

# d\_rehash

## Name

`d_rehash` — add an entry back to the hash

## Synopsis

```
void d_rehash (struct dentry * entry);
```

## Arguments

*entry*

dentry to add to the hash

## Description

Adds a dentry to the hash according to its name.

## d\_move

### Name

`d_move` — move a dentry

### Synopsis

```
void d_move (struct dentry * dentry, struct dentry * target);
```

### Arguments

*dentry*

entry to move

*target*

new dentry

### Description

Update the dcache to reflect the move of a file name. Negative dcache entries should not be moved in this way.

## **\_\_d\_path**

### **Name**

`__d_path` — return the path of a dentry

### **Synopsis**

```
char * __d_path (struct dentry * dentry, struct vfsmount *  
vfsmnt, struct dentry * root, struct vfsmount * rootmnt, char *  
buffer, int buflen);
```

### **Arguments**

*dentry*

dentry to report

*vfsmnt*

vfsmnt to which the dentry belongs

*root*

root dentry

*rootmnt*

vfsmnt to which the root dentry belongs

*buffer*

buffer to return value in

*buflen*

buffer length

## Description

Convert a dentry into an ASCII path name. If the entry has been deleted the string “(deleted)” is appended. Note that this is ambiguous. Returns the buffer.

“buflen” should be `PAGE_SIZE` or more. Caller holds the `dcache_lock`.

## is\_subdir

### Name

`is_subdir` — is new dentry a subdirectory of old\_dentry

### Synopsis

```
int is_subdir (struct dentry * new_dentry, struct dentry *  
old_dentry);
```

## Arguments

*new\_dentry*

new dentry

*old\_dentry*

old dentry

## Description

Returns 1 if *new\_dentry* is a subdirectory of the parent (at any depth). Returns 0 otherwise.

# find\_inode\_number

## Name

*find\_inode\_number* — check for dentry with name

## Synopsis

```
ino_t find_inode_number (struct dentry * dir, struct qstr *  
name);
```



## Arguments

*dir*

directory to check

*name*

Name to find.

## Description

Check whether a dentry already exists for the given name, and return the inode number if it has an inode. Otherwise 0 is returned.

This routine is used to post-process directory listings for filesystems using synthetic inode numbers, and is necessary to keep `getcwd` working.

## d\_drop

### Name

`d_drop` — drop a dentry

### Synopsis

```
void d_drop (struct dentry * dentry);
```

## Arguments

*dentry*

dentry to drop

## Description

`d_drop` unhashes the entry from the parent dentry hashes, so that it won't be found through a VFS lookup any more. Note that this is different from deleting the dentry - `d_delete` will try to mark the dentry negative if possible, giving a successful `_negative_` lookup, while `d_drop` will just make the cache lookup fail.

`d_drop` is used mainly for stuff that wants to invalidate a dentry for some reason (NFS timeouts or autofs deletes).

## `d_add`

### Name

`d_add` — add dentry to hash queues

## Synopsis

```
void d_add (struct dentry * entry, struct inode * inode);
```

## Arguments

*entry*

dentry to add

*inode*

The inode to attach to this dentry

## Description

This adds the entry to the hash queues and initializes *inode*. The entry was actually filled in earlier during `d_alloc`.

## dget

### Name

`dget` — get a reference to a dentry

## Synopsis

```
struct dentry * dget (struct dentry * dentry);
```

## Arguments

*dentry*

dentry to get a reference to

## Description

Given a dentry or `NULL` pointer increment the reference count if appropriate and return the dentry. A dentry will not be destroyed when it has references. `dget` should never be called for dentries with zero reference counter. For these cases (preferably none, functions in `dcache.c` are sufficient for normal needs and they take necessary precautions) you should hold `dcache_lock` and call `dget_locked` instead of `dget`.

## d\_unhashed

### Name

`d_unhashed` — is dentry hashed

## Synopsis

```
int d_unhashed (struct dentry * dentry);
```

## Arguments

*dentry*

entry to check

## Description

Returns true if the dentry passed is not currently hashed.

## 6.2. Inode Handling

### **\_\_mark\_inode\_dirty**

#### **Name**

`__mark_inode_dirty` — internal function

## Synopsis

```
void __mark_inode_dirty (struct inode * inode, int flags);
```

## Arguments

*inode*

inode to mark

*flags*

what kind of dirty (i.e. I\_DIRTY\_SYNC) Mark an inode as dirty. Callers should use mark\_inode\_dirty or mark\_inode\_dirty\_sync.

## write\_inode\_now

### Name

write\_inode\_now — write an inode to disk

## Synopsis

```
void write_inode_now (struct inode * inode, int sync);
```

## Arguments

*inode*

inode to write to disk

*sync*

whether the write should be synchronous or not

## Description

This function commits an inode to disk immediately if it is dirty. This is primarily needed by knfsd.

# clear\_inode

## Name

`clear_inode` — clear an inode

## Synopsis

```
void clear_inode (struct inode * inode);
```

## Arguments

*inode*

inode to clear

## Description

This is called by the filesystem to tell us that the inode is no longer useful. We just terminate it with extreme prejudice.

# invalidate\_inodes

## Name

`invalidate_inodes` — discard the inodes on a device

## Synopsis

```
int invalidate_inodes (struct super_block * sb);
```



## Arguments

*sb*

superblock

## Description

Discard all of the inodes for a given superblock. If the discard fails because there are busy inodes then a non zero value is returned. If the discard is successful all the inodes have been discarded.

# get\_empty\_inode

## Name

get\_empty\_inode — obtain an inode

## Synopsis

```
struct inode * get_empty_inode ( void );
```

## Arguments

*void*

no arguments

## Description

This is called by things like the networking layer etc that want to get an inode without any inode number, or filesystems that allocate new inodes with no pre-existing information.

On a successful return the inode pointer is returned. On a failure a `NULL` pointer is returned. The returned inode is not on any superblock lists.

# iunique

## Name

`iunique` — get a unique inode number

## Synopsis

```
ino_t iunique (struct super_block * sb, ino_t max_reserved);
```

## Arguments

*sb*

superblock

*max\_reserved*

highest reserved inode number

## Description

Obtain an inode number that is unique on the system for a given superblock. This is used by file systems that have no natural permanent inode numbering system. An inode number is returned that is higher than the reserved limit but unique.

## BUGS

With a large number of inodes live on the file system this function currently becomes quite slow.

# insert\_inode\_hash

## Name

`insert_inode_hash` — hash an inode

## Synopsis

```
void insert_inode_hash (struct inode * inode);
```

## Arguments

*inode*

unhashed inode

## Description

Add an inode to the inode hash for this superblock. If the inode has no superblock it is added to a separate anonymous chain.

# remove\_inode\_hash

## Name

remove\_inode\_hash — remove an inode from the hash

## Synopsis

```
void remove_inode_hash (struct inode * inode);
```

## Arguments

*inode*

inode to unhash

## Description

Remove an inode from the superblock or anonymous hash.

# iput

## Name

`iput` — put an inode

## Synopsis

```
void iput (struct inode * inode);
```

## Arguments

*inode*

inode to put

## Description

Puts an inode, dropping its usage count. If the inode use count hits zero the inode is also then freed and may be destroyed.

# bmap

## Name

bmap — find a block number in a file

## Synopsis

```
int bmap (struct inode * inode, int block);
```

## Arguments

*inode*

inode of file

*block*

block to find

## Description

Returns the block number on the device holding the inode that is the disk block number for the block of the file requested. That is, asked for block 4 of inode 1 the function will return the disk block relative to the disk start that holds that block of the file.

# update\_atime

## Name

`update_atime` — update the access time

## Synopsis

```
void update_atime (struct inode * inode);
```

## Arguments

*inode*

inode accessed

## Description

Update the accessed time on an inode and mark it for writeback. This function automatically handles read only file systems and media, as well as the “noatime” flag and inode specific “noatime” markers.

## make\_bad\_inode

### Name

`make_bad_inode` — mark an inode bad due to an I/O error

### Synopsis

```
void make_bad_inode (struct inode * inode);
```



## Arguments

*inode*

Inode to mark bad

## Description

When an inode cannot be read due to a media or remote network failure this function makes the inode “bad” and causes I/O operations on it to fail from this point on.

# is\_bad\_inode

## Name

`is_bad_inode` — is an inode errored

## Synopsis

```
int is_bad_inode (struct inode * inode);
```

## Arguments

*inode*

inode to test

## Description

Returns true if the inode in question has been marked as bad.

## 6.3. Registration and Superblocks

### register\_filesystem

#### Name

`register_filesystem` — register a new filesystem

#### Synopsis

```
int register_filesystem (struct file_system_type * fs);
```

## Arguments

*fs*

the file system structure

## Description

Adds the file system passed to the list of file systems the kernel is aware of for mount and other syscalls. Returns 0 on success, or a negative errno code on an error.

The &struct file\_system\_type that is passed is linked into the kernel structures and must not be freed until the file system has been unregistered.

# unregister\_filesystem

## Name

unregister\_filesystem — unregister a file system

## Synopsis

```
int unregister_filesystem (struct file_system_type * fs);
```

## Arguments

*fs*

filesystem to unregister

## Description

Remove a file system that was previously successfully registered with the kernel. An error is returned if the file system is not found. Zero is returned on a success.

Once this function has returned the `&struct file_system_type` structure may be freed or reused.

## get\_super

### Name

`get_super` — get the superblock of a device

### Synopsis

```
struct super_block * get_super (kdev_t dev);
```

## Arguments

*dev*

device to get the superblock for

## Description

Scans the superblock list and finds the superblock of the file system mounted on the device given. `NULL` is returned if no match is found.

## 6.4. File Locks

### `posix_lock_file`

#### Name

`posix_lock_file` —

#### Synopsis

```
int posix_lock_file (struct file * filp, struct file_lock *  
caller, unsigned int wait);
```

## Arguments

*filp*

The file to apply the lock to

*caller*

The lock to be applied

*wait*

1 to retry automatically, 0 to return -EAGAIN

## Description

Add a POSIX style lock to a file. We merge adjacent locks whenever possible. POSIX locks are sorted by owner task, then by starting address

## Kai Petzke writes

To make freeing a lock much faster, we keep a pointer to the lock before the actual one. But the real gain of the new coding was, that `lock_it` and `unlock_it` became one function.

## To all purists

Yes, I use a few `goto`'s. Just pass on to the next function.

## **\_\_get\_lease**

### **Name**

`__get_lease` — revoke all outstanding leases on file

### **Synopsis**

```
int __get_lease (struct inode * inode, unsigned int mode);
```

### **Arguments**

*inode*

the inode of the file to return

*mode*

the open mode (read or write)

### **Description**

`get_lease` (inlined for speed) has checked there already is a lease on this file. Leases are broken on a call to `open` or `truncate`. This function can sleep unless you specified `O_NONBLOCK` to your `open`.

# lease\_get\_mtime

## Name

`lease_get_mtime` —

## Synopsis

```
time_t lease_get_mtime (struct inode * inode);
```

## Arguments

*inode*

the inode

## Description

This is to force NFS clients to flush their caches for files with exclusive leases. The justification is that if someone has an exclusive lease, then they could be modifying it.



# posix\_block\_lock

## Name

`posix_block_lock` — blocks waiting for a file lock

## Synopsis

```
void posix_block_lock (struct file_lock * blocker, struct  
file_lock * waiter);
```

## Arguments

*blocker*

the lock which is blocking

*waiter*

the lock which conflicts and has to wait

## Description

lockd needs to block waiting for locks.

# posix\_unblock\_lock

## Name

`posix_unblock_lock` — stop waiting for a file lock

## Synopsis

```
void posix_unblock_lock (struct file_lock * waiter);
```

## Arguments

*waiter*

the lock which was waiting

## Description

lockd needs to block waiting for locks.

# lock\_may\_read

## Name

`lock_may_read` — checks that the region is free of locks

## Synopsis

```
int lock_may_read (struct inode * inode, loff_t start, unsigned  
long len);
```

## Arguments

*inode*

the inode that is being read

*start*

the first byte to read

*len*

the number of bytes to read

## Description

Emulates Windows locking requirements. Whole-file mandatory locks (share modes) can prohibit a read and byte-range POSIX locks can prohibit a read if they overlap.

N.B. this function is only ever called from knfsd and ownership of locks is never checked.

## lock\_may\_write

### Name

`lock_may_write` — checks that the region is free of locks

### Synopsis

```
int lock_may_write (struct inode * inode, loff_t start, unsigned
long len);
```

### Arguments

*inode*

the inode that is being written

*start*

the first byte to write

*len*

the number of bytes to write

## Description

Emulates Windows locking requirements. Whole-file mandatory locks (share modes) can prohibit a write and byte-range POSIX locks can prohibit a write if they overlap.

N.B. this function is only ever called from knfsd and ownership of locks is never checked.

## fcntl\_getlease

### Name

`fcntl_getlease` — Enquire what lease is currently active

### Synopsis

```
int fcntl_getlease (struct file * filp);
```

## Arguments

*filp*

the file

## Description

The value returned by this function will be one of

`F_RDLCK` to indicate a read-only (type II) lease is held.

`F_WRLCK` to indicate an exclusive lease is held.

## XXX

sfr & i disagree over whether `F_INPROGRESS` should be returned to userspace.

# fcntl\_setlease

## Name

`fcntl_setlease` — sets a lease on an open file

## Synopsis

```
int fcntl_setlease (unsigned int fd, struct file * filp, long  
arg);
```

## Arguments

*fd*

open file descriptor

*filp*

file pointer

*arg*

type of lease to obtain

## Description

Call this `fcntl` to establish a lease on the file. Note that you also need to call `F_SETSIG` to receive a signal when the lease is broken.

# sys\_flock

## Name

`sys_flock` — flock system call.

## Synopsis

```
asmlinkage long sys_flock (unsigned int fd, unsigned int cmd);
```

## Arguments

*fd*

the file descriptor to lock.

*cmd*

the type of lock to apply.

## Description

Apply a `FL_FLOCK` style lock to an open file descriptor. The *cmd* can be one of

`LOCK_SH` – a shared lock.

`LOCK_EX` – an exclusive lock.

`LOCK_UN` – remove an existing lock.



LOCK\_MAND – a ‘mandatory’ flock. This exists to emulate Windows Share Modes.

LOCK\_MAND can be combined with LOCK\_READ or LOCK\_WRITE to allow other processes read and write access respectively.

## get\_locks\_status

### Name

get\_locks\_status — reports lock usage in /proc/locks

### Synopsis

```
int get_locks_status (char * buffer, char ** start, off_t
offset, int length);
```

### Arguments

*buffer*

address in userspace to write into

*start*

?

*offset*

how far we are through the buffer

*length*

how much to read

# Chapter 7. Linux Networking

## 7.1. Socket Buffer Functions

### **skb\_queue\_empty**

#### **Name**

`skb_queue_empty` — check if a queue is empty

#### **Synopsis**

```
int skb_queue_empty (struct sk_buff_head * list);
```

#### **Arguments**

*list*

queue head

#### **Description**

Returns true if the queue is empty, false otherwise.

# skb\_get

## Name

skb\_get — reference buffer

## Synopsis

```
struct sk_buff * skb_get (struct sk_buff * skb);
```

## Arguments

*skb*

buffer to reference

## Description

Makes another reference to a socket buffer and returns a pointer to the buffer.

# kfree\_skb

## Name

kfree\_skb — free an sk\_buff

## Synopsis

```
void kfree_skb (struct sk_buff * skb);
```

## Arguments

*skb*

buffer to free

## Description

Drop a reference to the buffer and free it if the usage count has hit zero.

# skb\_cloned

## Name

`skb_cloned` — is the buffer a clone

## Synopsis

```
int skb_cloned (struct sk_buff * skb);
```

## Arguments

*skb*

buffer to check

## Description

Returns true if the buffer was generated with `skb_clone` and is one of multiple shared copies of the buffer. Cloned buffers are shared data so must not be written to under normal circumstances.

# skb\_shared

## Name

`skb_shared` — is the buffer shared

## Synopsis

```
int skb_shared (struct sk_buff * skb);
```

## Arguments

*skb*

buffer to check

## Description

Returns true if more than one person has a reference to this buffer.

# skb\_share\_check

## Name

`skb_share_check` — check if buffer is shared and if so clone it

## Synopsis

```
struct sk_buff * skb_share_check (struct sk_buff * skb, int  
pri);
```

## Arguments

*skb*

buffer to check

*pri*

priority for memory allocation

## Description

If the buffer is shared the buffer is cloned and the old copy drops a reference. A new clone with a single reference is returned. If the buffer is not shared the original buffer is returned. When being called from interrupt status or with spinlocks held *pri* must be GFP\_ATOMIC.



NULL is returned on a memory allocation failure.

## skb\_unshare

### Name

`skb_unshare` — make a copy of a shared buffer

### Synopsis

```
struct sk_buff * skb_unshare (struct sk_buff * skb, int pri);
```

### Arguments

*skb*

buffer to check

*pri*

priority for memory allocation

## Description

If the socket buffer is a clone then this function creates a new copy of the data, drops a reference count on the old copy and returns the new copy with the reference count at 1. If the buffer is not a clone the original buffer is returned. When called with a spinlock held or from interrupt state *pri* must be GFP\_ATOMIC

NULL is returned on a memory allocation failure.

## skb\_peek

### Name

skb\_peek —

### Synopsis

```
struct sk_buff * skb_peek (struct sk_buff_head * list_);
```

### Arguments

*list\_*

list to peek at

## Description

Peek an `&sk_buff`. Unlike most other operations you ***MUST*** be careful with this one. A peek leaves the buffer on the list and someone else may run off with it. You must hold the appropriate locks or have a private queue to do this.

Returns `NULL` for an empty list or a pointer to the head element. The reference count is not incremented and the reference is therefore volatile. Use with caution.

## `skb_peek_tail`

### Name

`skb_peek_tail` —

### Synopsis

```
struct sk_buff * skb_peek_tail (struct sk_buff_head * list_);
```

### Arguments

*list\_*

list to peek at

## Description

Peek an `&sk_buff`. Unlike most other operations you ***MUST*** be careful with this one. A peek leaves the buffer on the list and someone else may run off with it. You must hold the appropriate locks or have a private queue to do this.

Returns `NULL` for an empty list or a pointer to the tail element. The reference count is not incremented and the reference is therefore volatile. Use with caution.

## `skb_queue_len`

### Name

`skb_queue_len` — get queue length

### Synopsis

```
__u32 skb_queue_len (struct sk_buff_head * list_);
```

### Arguments

*list\_*

list to measure

## Description

Return the length of an `&sk_buff` queue.

# `__skb_queue_head`

## Name

`__skb_queue_head` — queue a buffer at the list head

## Synopsis

```
void __skb_queue_head (struct sk_buff_head * list, struct  
sk_buff * newsk);
```

## Arguments

*list*

list to use

*newsk*

buffer to queue

## Description

Queue a buffer at the start of a list. This function takes no locks and you must therefore hold required locks before calling it.

A buffer cannot be placed on two lists at the same time.

# skb\_queue\_head

## Name

`skb_queue_head` — queue a buffer at the list head

## Synopsis

```
void skb_queue_head (struct sk_buff_head * list, struct sk_buff  
* newsk);
```

## Arguments

*list*

list to use

*newsk*

buffer to queue

## Description

Queue a buffer at the start of the list. This function takes the list lock and can be used safely with other locking &sk\_buff functions safely.

A buffer cannot be placed on two lists at the same time.

## \_\_skb\_queue\_tail

### Name

`__skb_queue_tail` — queue a buffer at the list tail

### Synopsis

```
void __skb_queue_tail (struct sk_buff_head * list, struct  
sk_buff * newsk);
```

## Arguments

*list*

list to use

*newsk*

buffer to queue

## Description

Queue a buffer at the end of a list. This function takes no locks and you must therefore hold required locks before calling it.

A buffer cannot be placed on two lists at the same time.

# skb\_queue\_tail

## Name

`skb_queue_tail` — queue a buffer at the list tail

## Synopsis

```
void skb_queue_tail (struct sk_buff_head * list, struct sk_buff  
* newsk);
```



## Arguments

*list*

list to use

*newsk*

buffer to queue

## Description

Queue a buffer at the tail of the list. This function takes the list lock and can be used safely with other locking &sk\_buff functions safely.

A buffer cannot be placed on two lists at the same time.

## **\_\_skb\_dequeue**

### Name

`__skb_dequeue` — remove from the head of the queue

## Synopsis

```
struct sk_buff * __skb_dequeue (struct sk_buff_head * list);
```

## Arguments

*list*

list to dequeue from

## Description

Remove the head of the list. This function does not take any locks so must be used with appropriate locks held only. The head item is returned or `NULL` if the list is empty.

# skb\_dequeue

## Name

`skb_dequeue` — remove from the head of the queue

## Synopsis

```
struct sk_buff * skb_dequeue (struct sk_buff_head * list);
```

## Arguments

*list*

list to dequeue from

## Description

Remove the head of the list. The list lock is taken so the function may be used safely with other locking list functions. The head item is returned or NULL if the list is empty.

## skb\_insert

### Name

`skb_insert` — insert a buffer

## Synopsis

```
void skb_insert (struct sk_buff * old, struct sk_buff * newsk);
```

## Arguments

*old*

buffer to insert before

*newsk*

buffer to insert

## Description

Place a packet before a given packet in a list. The list locks are taken and this function is atomic with respect to other list locked calls. A buffer cannot be placed on two lists at the same time.

## skb\_append

### Name

`skb_append` — append a buffer

## Synopsis

```
void skb_append (struct sk_buff * old, struct sk_buff * newsk);
```

## Arguments

*old*

buffer to insert after

*newsk*

buffer to insert

## Description

Place a packet after a given packet in a list. The list locks are taken and this function is atomic with respect to other list locked calls. A buffer cannot be placed on two lists at the same time.

## skb\_unlink

### Name

`skb_unlink` — remove a buffer from a list

## Synopsis

```
void skb_unlink (struct sk_buff * skb);
```

## Arguments

*skb*

buffer to remove

## Description

Place a packet after a given packet in a list. The list locks are taken and this function is atomic with respect to other list locked calls

Works even without knowing the list it is sitting on, which can be handy at times. It also means that THE LIST MUST EXIST when you unlink. Thus a list must have its contents unlinked before it is destroyed.

## \_\_skb\_dequeue\_tail

### Name

`__skb_dequeue_tail` — remove from the tail of the queue

## Synopsis

```
struct sk_buff * __skb_dequeue_tail (struct sk_buff_head *  
list);
```

## Arguments

*list*

list to dequeue from

## Description

Remove the tail of the list. This function does not take any locks so must be used with appropriate locks held only. The tail item is returned or NULL if the list is empty.

## skb\_dequeue\_tail

### Name

`skb_dequeue_tail` — remove from the head of the queue

## Synopsis

```
struct sk_buff * skb_dequeue_tail (struct sk_buff_head * list);
```

## Arguments

*list*

list to dequeue from

## Description

Remove the head of the list. The list lock is taken so the function may be used safely with other locking list functions. The tail item is returned or `NULL` if the list is empty.

# skb\_put

## Name

`skb_put` — add data to a buffer



## Synopsis

```
unsigned char * skb_put (struct sk_buff * skb, unsigned int  
len);
```

## Arguments

*skb*

buffer to use

*len*

amount of data to add

## Description

This function extends the used data area of the buffer. If this would exceed the total buffer size the kernel will panic. A pointer to the first byte of the extra data is returned.

## skb\_push

### Name

`skb_push` — add data to the start of a buffer

## Synopsis

```
unsigned char * skb_push (struct sk_buff * skb, unsigned int  
len);
```

## Arguments

*skb*

buffer to use

*len*

amount of data to add

## Description

This function extends the used data area of the buffer at the buffer start. If this would exceed the total buffer headroom the kernel will panic. A pointer to the first byte of the extra data is returned.

# skb\_pull

## Name

`skb_pull` — remove data from the start of a buffer

## Synopsis

```
unsigned char * skb_pull (struct sk_buff * skb, unsigned int  
len);
```

## Arguments

*skb*

buffer to use

*len*

amount of data to remove

## Description

This function removes data from the start of a buffer, returning the memory to the headroom. A pointer to the next data in the buffer is returned. Once the data has been pulled future pushes will overwrite the old data.

# skb\_headroom

## Name

`skb_headroom` — bytes at buffer head

## Synopsis

```
int skb_headroom (const struct sk_buff * skb);
```

## Arguments

*skb*

buffer to check

## Description

Return the number of bytes of free space at the head of an `&sk_buff`.

# skb\_tailroom

## Name

`skb_tailroom` — bytes at buffer end

## Synopsis

```
int skb_tailroom (const struct sk_buff * skb);
```

## Arguments

*skb*

buffer to check

## Description

Return the number of bytes of free space at the tail of an `sk_buff`

# skb\_reserve

## Name

`skb_reserve` — adjust headroom

## Synopsis

```
void skb_reserve (struct sk_buff * skb, unsigned int len);
```

## Arguments

*skb*

buffer to alter

*len*

bytes to move

## Description

Increase the headroom of an empty `&sk_buff` by reducing the tail room. This is only allowed for an empty buffer.

# skb\_trim

## Name

`skb_trim` — remove end from a buffer

## Synopsis

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

## Arguments

*skb*

buffer to alter

*len*

new length

## Description

Cut the length of a buffer down by removing data from the tail. If the buffer is already under the length specified it is not modified.

# skb\_orphan

## Name

skb\_orphan — orphan a buffer

## Synopsis

```
void skb_orphan (struct sk_buff * skb);
```

## Arguments

*skb*

buffer to orphan

## Description

If a buffer currently has an owner then we call the owner's destructor function and make the *skb* unowned. The buffer continues to exist but is no longer charged to its former owner.



# skb\_queue\_purge

## Name

`skb_queue_purge` — empty a list

## Synopsis

```
void skb_queue_purge (struct sk_buff_head * list);
```

## Arguments

*list*

list to empty

## Description

Delete all buffers on an `&sk_buff` list. Each buffer is removed from the list and one reference dropped. This function takes the list lock and is atomic with respect to other list locking functions.

## **\_\_skb\_queue\_purge**

### **Name**

`__skb_queue_purge` — empty a list

### **Synopsis**

```
void __skb_queue_purge (struct sk_buff_head * list);
```

### **Arguments**

*list*

list to empty

### **Description**

Delete all buffers on an `&sk_buff` list. Each buffer is removed from the list and one reference dropped. This function does not take the list lock and the caller must hold the relevant locks to use it.

## **\_\_dev\_alloc\_skb**

### **Name**

`__dev_alloc_skb` — allocate an skbuff for sending

### **Synopsis**

```
struct sk_buff * __dev_alloc_skb (unsigned int length, int  
gfp_mask);
```

### **Arguments**

*length*

length to allocate

*gfp\_mask*

get\_free\_pages mask, passed to alloc\_skb

### **Description**

Allocate a new `&sk_buff` and assign it a usage count of one. The buffer has unspecified headroom built in. Users should allocate the headroom they think they need without accounting for the built in space. The built in space is used for optimisations.

NULL is returned in there is no free memory.

## dev\_alloc\_skb

### Name

`dev_alloc_skb` — allocate an skbuff for sending

### Synopsis

```
struct sk_buff * dev_alloc_skb (unsigned int length);
```

### Arguments

*length*

length to allocate

### Description

Allocate a new `&sk_buff` and assign it a usage count of one. The buffer has unspecified headroom built in. Users should allocate the headroom they think they need without accounting for the built in space. The built in space is used for optimisations.

`NULL` is returned if there is no free memory. Although this function allocates memory it can be called from an interrupt.

## skb\_cow

### Name

`skb_cow` — copy header of `skb` when it is required

### Synopsis

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

### Arguments

*skb*

buffer to cow

*headroom*

needed headroom

### Description

If the `skb` passed lacks sufficient headroom or its data part is shared, data is reallocated. If reallocation fails, an error is returned and original `skb` is not changed.

The result is `skb` with writable area `skb->head...skb->tail` and at least *headroom* of space at head.

## skb\_over\_panic

### Name

`skb_over_panic` — private function

### Synopsis

```
void skb_over_panic (struct sk_buff * skb, int sz, void * here);
```

### Arguments

*skb*

buffer

*sz*

size

*here*

address

## Description

Out of line support code for `skb_put`. Not user callable.

# skb\_under\_panic

## Name

`skb_under_panic` — private function

## Synopsis

```
void skb_under_panic (struct sk_buff * skb, int sz, void *  
here);
```

## Arguments

*skb*

buffer

*sz*

size

*here*

address

## Description

Out of line support code for `skb_push`. Not user callable.

# alloc\_skb

## Name

`alloc_skb` — allocate a network buffer

## Synopsis

```
struct sk_buff * alloc_skb (unsigned int size, int gfp_mask);
```

## Arguments

*size*

size to allocate



*gfp\_mask*

allocation mask

## Description

Allocate a new `&sk_buff`. The returned buffer has no headroom and a tail room of size bytes. The object has a reference count of one. The return is the buffer. On a failure the return is `NULL`.

Buffers may only be allocated from interrupts using a *gfp\_mask* of `GFP_ATOMIC`.

## `__kfree_skb`

### Name

`__kfree_skb` — private function

### Synopsis

```
void __kfree_skb (struct sk_buff * skb);
```

## Arguments

*skb*

buffer

## Description

Free an `sk_buff`. Release anything attached to the buffer. Clean the state. This is an internal helper function. Users should always call `kfree_skb`

# skb\_clone

## Name

`skb_clone` — duplicate an `sk_buff`

## Synopsis

```
struct sk_buff * skb_clone (struct sk_buff * skb, int gfp_mask);
```

## Arguments

*skb*

buffer to clone

*gfp\_mask*

allocation priority

## Description

Duplicate an `&sk_buff`. The new one is not owned by a socket. Both copies share the same packet data but not structure. The new buffer has a reference count of 1. If the allocation fails the function returns `NULL` otherwise the new buffer is returned.

If this function is called from an interrupt `gfp_mask` must be `GFP_ATOMIC`.

# skb\_copy

## Name

`skb_copy` — create private copy of an `sk_buff`

## Synopsis

```
struct sk_buff * skb_copy (const struct sk_buff * skb, int  
gfp_mask);
```

## Arguments

*skb*

buffer to copy

*gfp\_mask*

allocation priority

## Description

Make a copy of both an `&sk_buff` and its data. This is used when the caller wishes to modify the data and needs a private copy of the data to alter. Returns `NULL` on failure or the pointer to the buffer on success. The returned buffer has a reference count of 1.

As by-product this function converts non-linear `&sk_buff` to linear one, so that `&sk_buff` becomes completely private and caller is allowed to modify all the data of returned buffer. This means that this function is not recommended for use in circumstances when only header is going to be modified. Use `pskb_copy` instead.

## **pskb\_copy**

### Name

`pskb_copy` — create copy of an `sk_buff` with private head.

## Synopsis

```
struct sk_buff * pskb_copy (struct sk_buff * skb, int gfp_mask);
```

## Arguments

*skb*

buffer to copy

*gfp\_mask*

allocation priority

## Description

Make a copy of both an `&sk_buff` and part of its data, located in header. Fragmented data remain shared. This is used when the caller wishes to modify only header of `&sk_buff` and needs private copy of the header to alter. Returns `NULL` on failure or the pointer to the buffer on success. The returned buffer has a reference count of 1.

# pskb\_expand\_head

## Name

`pskb_expand_head` — reallocate header of `sk_buff`

## Synopsis

```
int pskb_expand_head (struct sk_buff * skb, int nhead, int  
ntail, int gfp_mask);
```

## Arguments

*skb*

buffer to reallocate

*nhead*

room to add at head

*ntail*

room to add at tail

*gfp\_mask*

allocation priority

## Description

Expands (or creates identical copy, if `&nhead` and `&ntail` are zero) header of `skb`. `&sk_buff` itself is not changed. `&sk_buff` MUST have reference count of 1. Returns zero in the case of success or error, if expansion failed. In the last case, `&sk_buff` is not changed.

All the pointers pointing into `skb` header may change and must be reloaded after call to this function.

## skb\_copy\_expand

### Name

`skb_copy_expand` — copy and expand `sk_buff`

### Synopsis

```
struct sk_buff * skb_copy_expand (const struct sk_buff * skb,  
int newheadroom, int newtailroom, int gfp_mask);
```

## Arguments

*skb*

buffer to copy

*newheadroom*

new free bytes at head

*newtailroom*

new free bytes at tail

*gfp\_mask*

allocation priority

## Description

Make a copy of both an `&sk_buff` and its data and while doing so allocate additional space.

This is used when the caller wishes to modify the data and needs a private copy of the data to alter as well as more space for new fields. Returns `NULL` on failure or the pointer to the buffer on success. The returned buffer has a reference count of 1.

You must pass `GFP_ATOMIC` as the allocation priority if this function is called from an interrupt.



# **\_\_pskb\_pull\_tail**

## **Name**

`__pskb_pull_tail` — advance tail of skb header

## **Synopsis**

```
unsigned char * __pskb_pull_tail (struct sk_buff * skb, int  
delta);
```

## **Arguments**

*skb*

buffer to reallocate

*delta*

number of bytes to advance tail

## **Description**

The function makes a sense only on a fragmented `&sk_buff`, it expands header moving its tail forward and copying necessary data from fragmented part.

`&sk_buff` MUST have reference count of 1.

Returns `NULL` (and `&sk_buff` does not change) if pull failed or value of new tail of `skb` in the case of success.

All the pointers pointing into `skb` header may change and must be reloaded after call to this function.

## 7.2. Socket Filter

### `sk_run_filter`

#### Name

`sk_run_filter` — run a filter on a socket

#### Synopsis

```
int sk_run_filter (struct sk_buff * skb, struct sock_filter *  
filter, int flen);
```

## Arguments

*skb*

buffer to run the filter on

*filter*

filter to apply

*flen*

length of filter

## Description

Decode and apply filter instructions to the `skb->data`. Return length to keep, 0 for none. `skb` is the data we are filtering, `filter` is the array of filter instructions, and `len` is the number of filter blocks in the array.

## sk\_chk\_filter

### Name

`sk_chk_filter` — verify socket filter code

## Synopsis

```
int sk_chk_filter (struct sock_filter * filter, int flen);
```

## Arguments

*filter*

filter to verify

*flen*

length of filter

## Description

Check the user's filter code. If we let some ugly filter code slip through kaboom! The filter must contain no references or jumps that are out of range, no illegal instructions and no backward jumps. It must end with a RET instruction

Returns 0 if the rule set is legal or a negative errno code if not.

# Chapter 8. Network device support

## 8.1. Driver Support

### **init\_etherdev**

#### **Name**

`init_etherdev` — Register ethernet device

#### **Synopsis**

```
struct net_device * init_etherdev (struct net_device * dev, int  
sizeof_priv);
```

#### **Arguments**

*dev*

An ethernet device structure to be filled in, or `NULL` if a new struct should be allocated.

*sizeof\_priv*

Size of additional driver-private structure to be allocated for this ethernet device

## Description

Fill in the fields of the device structure with ethernet-generic values.

If no device structure is passed, a new one is constructed, complete with a private data area of size *sizeof\_priv*. A 32-byte (not bit) alignment is enforced for this private data area.

If an empty string area is passed as dev->name, or a new structure is made, a new name string is constructed.

## alloc\_etherdev

### Name

`alloc_etherdev` — Allocates and sets up an ethernet device

### Synopsis

```
struct net_device * alloc_etherdev (int sizeof_priv);
```

### Arguments

*sizeof\_priv*

Size of additional driver-private structure to be allocated for this ethernet device

## Description

Fill in the fields of the device structure with ethernet-generic values. Basically does everything except registering the device.

Constructs a new net device, complete with a private data area of size *sizeof\_priv*. A 32-byte (not bit) alignment is enforced for this private data area.

## init\_fddidev

### Name

`init_fddidev` — Register FDDI device

### Synopsis

```
struct net_device * init_fddidev (struct net_device * dev, int
sizeof_priv);
```

### Arguments

*dev*

A FDDI device structure to be filled in, or NULL if a new struct should be allocated.

*sizeof\_priv*

Size of additional driver-private structure to be allocated for this ethernet device

## Description

Fill in the fields of the device structure with FDDI-generic values.

If no device structure is passed, a new one is constructed, complete with a private data area of size *sizeof\_priv*. A 32-byte (not bit) alignment is enforced for this private data area.

If an empty string area is passed as *dev->name*, or a new structure is made, a new name string is constructed.

## alloc\_fddidev

### Name

`alloc_fddidev` — Register FDDI device

### Synopsis

```
struct net_device * alloc_fddidev (int sizeof_priv);
```



## Arguments

*sizeof\_priv*

Size of additional driver-private structure to be allocated for this FDDI device

## Description

Fill in the fields of the device structure with FDDI-generic values.

Constructs a new net device, complete with a private data area of size *sizeof\_priv*. A 32-byte (not bit) alignment is enforced for this private data area.

# init\_hippi\_dev

## Name

`init_hippi_dev` — Register HIPPI device

## Synopsis

```
struct net_device * init_hippi_dev (struct net_device * dev, int
sizeof_priv);
```

## Arguments

*dev*

A HIPPI device structure to be filled in, or `NULL` if a new struct should be allocated.

*sizeof\_priv*

Size of additional driver-private structure to be allocated for this ethernet device

## Description

Fill in the fields of the device structure with HIPPI-generic values.

If no device structure is passed, a new one is constructed, complete with a private data area of size *sizeof\_priv*. A 32-byte (not bit) alignment is enforced for this private data area.

If an empty string area is passed as `dev->name`, or a new structure is made, a new name string is constructed.

## **alloc\_hippi\_dev**

### **Name**

`alloc_hippi_dev` — Register HIPPI device

## Synopsis

```
struct net_device * alloc_hippi_dev (int sizeof_priv);
```

## Arguments

*sizeof\_priv*

Size of additional driver-private structure to be allocated for this HIPPI device

## Description

Fill in the fields of the device structure with HIPPI-generic values.

Constructs a new net device, complete with a private data area of size *sizeof\_priv*. A 32-byte (not bit) alignment is enforced for this private data area.

## init\_trdev

### Name

`init_trdev` — Register token ring device

## Synopsis

```
struct net_device * init_trdev (struct net_device * dev, int  
sizeof_priv);
```

## Arguments

*dev*

A token ring device structure to be filled in, or `NULL` if a new struct should be allocated.

*sizeof\_priv*

Size of additional driver-private structure to be allocated for this ethernet device

## Description

Fill in the fields of the device structure with token ring-generic values.

If no device structure is passed, a new one is constructed, complete with a private data area of size *sizeof\_priv*. A 32-byte (not bit) alignment is enforced for this private data area.

If an empty string area is passed as *dev->name*, or a new structure is made, a new name string is constructed.

## alloc\_trdev

### Name

`alloc_trdev` — Register token ring device

### Synopsis

```
struct net_device * alloc_trdev (int sizeof_priv);
```

### Arguments

*sizeof\_priv*

Size of additional driver-private structure to be allocated for this token ring device

### Description

Fill in the fields of the device structure with token ring-generic values.

Constructs a new net device, complete with a private data area of size *sizeof\_priv*. A 32-byte (not bit) alignment is enforced for this private data area.

# init\_fcdev

## Name

`init_fcdev` — Register fibre channel device

## Synopsis

```
struct net_device * init_fcdev (struct net_device * dev, int
sizeof_priv);
```

## Arguments

*dev*

A fibre channel device structure to be filled in, or `NULL` if a new struct should be allocated.

*sizeof\_priv*

Size of additional driver-private structure to be allocated for this ethernet device

## Description

Fill in the fields of the device structure with fibre channel-generic values.

If no device structure is passed, a new one is constructed, complete with a private data area of size *sizeof\_priv*. A 32-byte (not bit) alignment is enforced for this private data area.

If an empty string area is passed as dev->name, or a new structure is made, a new name string is constructed.

## alloc\_fcdev

### Name

`alloc_fcdev` — Register fibre channel device

### Synopsis

```
struct net_device * alloc_fcdev (int sizeof_priv);
```

### Arguments

*sizeof\_priv*

Size of additional driver-private structure to be allocated for this fibre channel device

## Description

Fill in the fields of the device structure with fibre channel-generic values.

Constructs a new net device, complete with a private data area of size *sizeof\_priv*.  
A 32-byte (not bit) alignment is enforced for this private data area.

## dev\_add\_pack

### Name

`dev_add_pack` — add packet handler

### Synopsis

```
void dev_add_pack (struct packet_type * pt);
```

### Arguments

*pt*

packet type declaration



## Description

Add a protocol handler to the networking stack. The passed `&packet_type` is linked into kernel lists and may not be freed until it has been removed from the kernel lists.

# dev\_remove\_pack

## Name

`dev_remove_pack` — remove packet handler

## Synopsis

```
void dev_remove_pack (struct packet_type * pt);
```

## Arguments

*pt*

packet type declaration

## Description

Remove a protocol handler that was previously added to the kernel protocol handlers by `dev_add_pack`. The passed `&packet_type` is removed from the kernel lists and can be freed or reused once this function returns.

## `__dev_get_by_name`

### Name

`__dev_get_by_name` — find a device by its name

### Synopsis

```
struct net_device * __dev_get_by_name (const char * name);
```

### Arguments

*name*

name to find

## Description

Find an interface by name. Must be called under RTNL semaphore or *dev\_base\_lock*. If the name is found a pointer to the device is returned. If the name is not found then `NULL` is returned. The reference counters are not incremented so the caller must be careful with locks.

## dev\_get\_by\_name

### Name

`dev_get_by_name` — find a device by its name

### Synopsis

```
struct net_device * dev_get_by_name (const char * name);
```

### Arguments

*name*

name to find

## Description

Find an interface by name. This can be called from any context and does its own locking. The returned handle has the usage count incremented and the caller must use `dev_put` to release it when it is no longer needed. `NULL` is returned if no matching device is found.

## dev\_get

### Name

`dev_get` — test if a device exists

### Synopsis

```
int dev_get (const char * name);
```

### Arguments

*name*

name to test for

## Description

Test if a name exists. Returns true if the name is found. In order to be sure the name is not allocated or removed during the test the caller must hold the rtnl semaphore.

This function primarily exists for back compatibility with older drivers.

## **\_\_dev\_get\_by\_index**

### Name

`__dev_get_by_index` — find a device by its ifindex

### Synopsis

```
struct net_device * __dev_get_by_index (int ifindex);
```

### Arguments

*ifindex*

index of device

## Description

Search for an interface by index. Returns `NULL` if the device is not found or a pointer to the device. The device has not had its reference counter increased so the caller must be careful about locking. The caller must hold either the RTNL semaphore or *dev\_base\_lock*.

## dev\_get\_by\_index

### Name

`dev_get_by_index` — find a device by its ifindex

### Synopsis

```
struct net_device * dev_get_by_index (int ifindex);
```

### Arguments

*ifindex*

index of device

## Description

Search for an interface by index. Returns NULL if the device is not found or a pointer to the device. The device returned has had a reference added and the pointer is safe until the user calls `dev_put` to indicate they have finished with it.

## `dev_alloc_name`

### Name

`dev_alloc_name` — allocate a name for a device

### Synopsis

```
int dev_alloc_name (struct net_device * dev, const char * name);
```

### Arguments

*dev*

device

*name*

name format string

## Description

Passed a format string - eg "ltd" it will try and find a suitable id. Not efficient for many devices, not called a lot. The caller must hold the dev\_base or rtnl lock while allocating the name and adding the device in order to avoid duplicates. Returns the number of the unit assigned or a negative errno code.

## dev\_alloc

### Name

`dev_alloc` — allocate a network device and name

### Synopsis

```
struct net_device * dev_alloc (const char * name, int * err);
```

### Arguments

*name*

name format string

*err*

error return pointer



## Description

Passed a format string, eg. "ltd", it will allocate a network device and space for the name. `NULL` is returned if no memory is available. If the allocation succeeds then the name is assigned and the device pointer returned. `NULL` is returned if the name allocation failed. The cause of an error is returned as a negative `errno` code in the variable `err` points to.

The caller must hold the `dev_base` or RTNL locks when doing this in order to avoid duplicate name allocations.

## netdev\_state\_change

### Name

`netdev_state_change` — device changes state

### Synopsis

```
void netdev_state_change (struct net_device * dev);
```

## Arguments

*dev*

device to cause notification

## Description

Called to indicate a device has changed state. This function calls the notifier chains for `netdev_chain` and sends a `NEWLINK` message to the routing socket.

## `dev_load`

### Name

`dev_load` — load a network module

## Synopsis

```
void dev_load (const char * name);
```

## Arguments

*name*

name of interface

## Description

If a network interface is not present and the process has suitable privileges this function loads the module. If module loading is not available in this kernel then it becomes a nop.

## dev\_open

### Name

dev\_open — prepare an interface for use.

### Synopsis

```
int dev_open (struct net_device * dev);
```

## Arguments

*dev*

device to open

## Description

Takes a device from down to up state. The device's private open function is invoked and then the multicast lists are loaded. Finally the device is moved into the up state and a `NETDEV_UP` message is sent to the netdev notifier chain.

Calling this function on an active interface is a nop. On a failure a negative errno code is returned.

## dev\_close

### Name

`dev_close` — shutdown an interface.

### Synopsis

```
int dev_close (struct net_device * dev);
```

## Arguments

*dev*

device to shutdown

## Description

This function moves an active device into down state. A `NETDEV_GOING_DOWN` is sent to the netdev notifier chain. The device is then deactivated and finally a `NETDEV_DOWN` is sent to the notifier chain.

# register\_netdevice\_notifier

## Name

`register_netdevice_notifier` — register a network notifier block

## Synopsis

```
int register_netdevice_notifier (struct notifier_block * nb);
```

## Arguments

*nb*

notifier

## Description

Register a notifier to be called when network device events occur. The notifier passed is linked into the kernel structures and must not be reused until it has been unregistered. A negative errno code is returned on a failure.

# unregister\_netdevice\_notifier

## Name

`unregister_netdevice_notifier` — unregister a network notifier block

## Synopsis

```
int unregister_netdevice_notifier (struct notifier_block * nb);
```

## Arguments

*nb*

notifier

## Description

Unregister a notifier previously registered by `register_netdevice_notifier`. The notifier is unlinked into the kernel structures and may then be reused. A negative errno code is returned on a failure.

## dev\_queue\_xmit

### Name

`dev_queue_xmit` — transmit a buffer

### Synopsis

```
int dev_queue_xmit (struct sk_buff * skb);
```

## Arguments

*skb*

buffer to transmit

## Description

Queue a buffer for transmission to a network device. The caller must have set the device and priority and built the buffer before calling this function. The function can be called from an interrupt.

A negative errno code is returned on a failure. A success does not guarantee the frame will be transmitted as it may be dropped due to congestion or traffic shaping.

## netif\_rx

### Name

`netif_rx` — post buffer to the network code

### Synopsis

```
int netif_rx (struct sk_buff * skb);
```



## Arguments

*skb*

buffer to post

## Description

This function receives a packet from a device driver and queues it for the upper (protocol) levels to process. It always succeeds. The buffer may be dropped during processing for congestion control or by the protocol layers.

## return values

NET\_RX\_SUCCESS (no congestion) NET\_RX\_CN\_LOW (low congestion)

NET\_RX\_CN\_MOD (moderate congestion) NET\_RX\_CN\_HIGH (high congestion)

NET\_RX\_DROP (packet was dropped)

# net\_call\_rx\_atomic

## Name

net\_call\_rx\_atomic —

## Synopsis

```
void net_call_rx_atomic (void (*fn) (void));
```

## Arguments

*fn*

function to call

## Description

Make a function call that is atomic with respect to the protocol layers.

# register\_gifconf

## Name

register\_gifconf — register a SIOCGIF handler

## Synopsis

```
int register_gifconf (unsigned int family, gifconf_func_t *  
gifconf);
```

## Arguments

*family*

Address family

*gifconf*

Function handler

## Description

Register protocol dependent address dumping routines. The handler that is passed must not be freed or reused until it has been replaced by another handler.

## netdev\_set\_master

### Name

`netdev_set_master` — set up master/slave pair

## Synopsis

```
int netdev_set_master (struct net_device * slave, struct
net_device * master);
```

## Arguments

*slave*

slave device

*master*

new master device

## Description

Changes the master device of the slave. Pass `NULL` to break the bonding. The caller must hold the RTNL semaphore. On a failure a negative `errno` code is returned. On success the reference counts are adjusted, `RTM_NEWLINK` is sent to the routing socket and the function returns zero.

## dev\_set\_promiscuity

### Name

`dev_set_promiscuity` — update promiscuity count on a device

### Synopsis

```
void dev_set_promiscuity (struct net_device * dev, int inc);
```

### Arguments

*dev*

device

*inc*

modifier

### Description

Add or remove promiscuity from a device. While the count in the device remains above zero the interface remains promiscuous. Once it hits zero the device reverts back to normal filtering operation. A negative *inc* value is used to drop promiscuity on the device.

## dev\_set\_allmulti

### Name

`dev_set_allmulti` — update allmulti count on a device

### Synopsis

```
void dev_set_allmulti (struct net_device * dev, int inc);
```

### Arguments

*dev*

device

*inc*

modifier

### Description

Add or remove reception of all multicast frames to a device. While the count in the device remains above zero the interface remains listening to all interfaces. Once it hits

zero the device reverts back to normal filtering operation. A negative *inc* value is used to drop the counter when releasing a resource needing all multicasts.

## dev\_ioctl

### Name

`dev_ioctl` — network device ioctl

### Synopsis

```
int dev_ioctl (unsigned int cmd, void * arg);
```

### Arguments

*cmd*

command to issue

*arg*

pointer to a struct ifreq in user space

## Description

Issue ioctl functions to devices. This is normally called by the user space syscall interfaces but can sometimes be useful for other purposes. The return value is the return from the syscall if positive or a negative errno code on error.

# dev\_new\_index

## Name

`dev_new_index` — allocate an ifindex

## Synopsis

```
int dev_new_index ( void );
```

## Arguments

*void*

no arguments

## Description



Returns a suitable unique value for a new device interface number. The caller must hold the `rtnl` semaphore or the `dev_base_lock` to be sure it remains unique.

## netdev\_finish\_unregister

### Name

`netdev_finish_unregister` — complete unregistration

### Synopsis

```
int netdev_finish_unregister (struct net_device * dev);
```

### Arguments

*dev*

device

### Description

Destroy and free a dead device. A value of zero is returned on success.

# unregister\_netdevice

## Name

`unregister_netdevice` — remove device from the kernel

## Synopsis

```
int unregister_netdevice (struct net_device * dev);
```

## Arguments

*dev*

device

## Description

This function shuts down a device interface and removes it from the kernel tables. On success 0 is returned, on a failure a negative `errno` code is returned.

Callers must hold the `rtnl` semaphore. See the comment at the end of `Space.c` for details about the locking. You may want `unregister_netdev` instead of this.

## 8.2. 8390 Based Network Cards

### ei\_open

#### Name

`ei_open` — Open/initialize the board.

#### Synopsis

```
int ei_open (struct net_device * dev);
```

#### Arguments

*dev*

network device to initialize

## Description

This routine goes all-out, setting everything up anew at each open, even though many of these registers should only need to be set once at boot.

## ei\_close

### Name

`ei_close` — shut down network device

### Synopsis

```
int ei_close (struct net_device * dev);
```

### Arguments

*dev*

network device to close

### Description

Opposite of `ei_open`. Only used when “`ifconfig <devname> down`” is done.

## ei\_tx\_timeout

### Name

`ei_tx_timeout` — handle transmit time out condition

### Synopsis

```
void ei_tx_timeout (struct net_device * dev);
```

### Arguments

*dev*

network device which has apparently fallen asleep

### Description

Called by kernel when device never acknowledges a transmit has completed (or failed)  
- i.e. never posted a Tx related interrupt.

## ei\_interrupt

### Name

`ei_interrupt` — handle the interrupts from an 8390

### Synopsis

```
void ei_interrupt (int irq, void * dev_id, struct pt_regs *  
regs);
```

### Arguments

*irq*

interrupt number

*dev\_id*

a pointer to the `net_device`

*regs*

unused

### Description

Handle the ether interface interrupts. We pull packets from the 8390 via the card specific functions and fire them at the networking stack. We also handle transmit

completions and wake the transmit path if neccessary. We also update the counters and do other housekeeping as needed.

## ethdev\_init

### Name

`ethdev_init` — init rest of 8390 device struct

### Synopsis

```
int ethdev_init (struct net_device * dev);
```

### Arguments

*dev*

network device structure to init

### Description

Initialize the rest of the 8390 device structure. Do NOT `__init` this, as it is used by 8390 based modular drivers too.

## NS8390\_init

### Name

`NS8390_init` — initialize 8390 hardware

### Synopsis

```
void NS8390_init (struct net_device * dev, int startp);
```

### Arguments

*dev*

network device to initialize

*startp*

boolean. non-zero value to initiate chip processing

### Description

Must be called with lock held.



## 8.3. Synchronous PPP

### sppp\_input

#### Name

`sppp_input` — receive and process a WAN PPP frame

#### Synopsis

```
void sppp_input (struct net_device * dev, struct sk_buff * skb);
```

#### Arguments

*dev*

The device it arrived on

*skb*

The buffer to process

## Description

This can be called directly by cards that do not have timing constraints but is normally called from the network layer after interrupt servicing to process frames queued via `netif_rx`.

We process the options in the card. If the frame is destined for the protocol stacks then it queues the frame for the upper level protocol. If it is a control from it is processed and discarded here.

## sppp\_close

### Name

`sppp_close` — close down a synchronous PPP or Cisco HDLC link

### Synopsis

```
int sppp_close (struct net_device * dev);
```

### Arguments

*dev*

The network device to drop the link of

## Description

This drops the logical interface to the channel. It is not done politely as we assume we will also be dropping DTR. Any timeouts are killed.

## sppp\_open

### Name

sppp\_open — open a synchronous PPP or Cisco HDLC link

### Synopsis

```
int sppp_open (struct net_device * dev);
```

### Arguments

*dev*

Network device to activate

## Description

Close down any existing synchronous session and commence from scratch. In the PPP case this means negotiating LCP/IPCPC and friends, while for Cisco HDLC we simply need to start sending keepalives

## sppp\_reopen

### Name

`sppp_reopen` — notify of physical link loss

### Synopsis

```
int sppp_reopen (struct net_device * dev);
```

### Arguments

*dev*

Device that lost the link

## Description

This function informs the synchronous protocol code that the underlying link died (for example a carrier drop on X.21)

We increment the magic numbers to ensure that if the other end failed to notice we will correctly start a new session. It happens do to the nature of telco circuits is that you can lose carrier on one end only.

Having done this we go back to negotiating. This function may be called from an interrupt context.

## sppp\_change\_mtu

### Name

`sppp_change_mtu` — Change the link MTU

### Synopsis

```
int sppp_change_mtu (struct net_device * dev, int new_mtu);
```

## Arguments

*dev*

Device to change MTU on

*new\_mtu*

New MTU

## Description

Change the MTU on the link. This can only be called with the link down. It returns an error if the link is up or the mtu is out of range.

# sppp\_do\_ioctl

## Name

sppp\_do\_ioctl — Ioctl handler for ppp/hdlc

## Synopsis

```
int sppp_do_ioctl (struct net_device * dev, struct ifreq * ifr,  
int cmd);
```

## Arguments

*dev*

Device subject to ioctl

*ifr*

Interface request block from the user

*cmd*

Command that is being issued

## Description

This function handles the ioctls that may be issued by the user to control the settings of a PPP/HDLC link. It does both busy and security checks. This function is intended to be wrapped by callers who wish to add additional ioctl calls of their own.

# sppp\_attach

## Name

`sppp_attach` — attach synchronous PPP/HDLC to a device

## Synopsis

```
void sppp_attach (struct ppp_device * pd);
```

## Arguments

*pd*

PPP device to initialise

## Description

This initialises the PPP/HDLC support on an interface. At the time of calling the dev element must point to the network device that this interface is attached to. The interface should not yet be registered.

## **sppp\_detach**

### Name

`sppp_detach` — release PPP resources from a device



## Synopsis

```
void sppp_detach (struct net_device * dev);
```

## Arguments

*dev*

Network device to release

## Description

Stop and free up any PPP/HDLC resources used by this interface. This must be called before the device is freed.

# Chapter 9. Module Support

## 9.1. Module Loading

### **request\_module**

#### **Name**

`request_module` — try to load a kernel module

#### **Synopsis**

```
int request_module (const char * module_name);
```

#### **Arguments**

*module\_name*

Name of module

## Description

Load a module using the user mode module loader. The function returns zero on success or a negative `errno` code on failure. Note that a successful module load does not mean the module did not then unload and exit on an error of its own. Callers must check that the service they requested is now available not blindly invoke it.

If module auto-loading support is disabled then this function becomes a no-operation.

# call\_usermodehelper

## Name

`call_usermodehelper` — start a usermode application

## Synopsis

```
int call_usermodehelper (char * path, char ** argv, char **  
envp);
```

## Arguments

*path*

pathname for the application

*argv*

null-terminated argument list

*envp*

null-terminated environment list

## Description

Runs a user-space application. The application is started asynchronously. It runs as a child of keventd. It runs with full root capabilities. keventd silently reaps the child when it exits.

Must be called from process context. Returns zero on success, else a negative error code.

## 9.2. Inter Module support

### inter\_module\_register

#### Name

`inter_module_register` — register a new set of inter module data.

## Synopsis

```
void inter_module_register (const char * im_name, struct module  
* owner, const void * userdata);
```

## Arguments

*im\_name*

an arbitrary string to identify the data, must be unique

*owner*

module that is registering the data, always use THIS\_MODULE

*userdata*

pointer to arbitrary userdata to be registered

## Description

Check that the *im\_name* has not already been registered, complain if it has. For new data, add it to the *inter\_module\_entry* list.

# inter\_module\_unregister

## Name

`inter_module_unregister` — unregister a set of inter module data.

## Synopsis

```
void inter_module_unregister (const char * im_name);
```

## Arguments

*im\_name*

an arbitrary string to identify the data, must be unique

## Description

Check that the `im_name` has been registered, complain if it has not. For existing data, remove it from the `inter_module_entry` list.

# inter\_module\_get

## Name

`inter_module_get` — return arbitrary userdata from another module.

## Synopsis

```
const void * inter_module_get (const char * im_name);
```

## Arguments

*im\_name*

an arbitrary string to identify the data, must be unique

## Description

If the `im_name` has not been registered, return NULL. Try to increment the use count on the owning module, if that fails then return NULL. Otherwise return the userdata.

# inter\_module\_get\_request

## Name

`inter_module_get_request` — im get with automatic request\_module.

## Synopsis

```
const void * inter_module_get_request (const char * im_name,  
const char * modname);
```

## Arguments

*im\_name*

an arbitrary string to identify the data, must be unique

*modname*

module that is expected to register *im\_name*

## Description

If `inter_module_get` fails, do `request_module` then retry.



# inter\_module\_put

## Name

`inter_module_put` — release use of data from another module.

## Synopsis

```
void inter_module_put (const char * im_name);
```

## Arguments

*im\_name*

an arbitrary string to identify the data, must be unique

## Description

If the `im_name` has not been registered, complain, otherwise decrement the use count on the owning module.

# Chapter 10. Hardware Interfaces

## 10.1. Interrupt Handling

### **disable\_irq\_nosync**

#### **Name**

`disable_irq_nosync` — disable an irq without waiting

#### **Synopsis**

```
void disable_irq_nosync (unsigned int irq);
```

#### **Arguments**

*irq*

Interrupt to disable

## Description

Disable the selected interrupt line. Disables and Enables are nested. Unlike `disable_irq`, this function does not ensure existing instances of the IRQ handler have completed before returning.

This function may be called from IRQ context.

## `disable_irq`

### Name

`disable_irq` — disable an irq and wait for completion

### Synopsis

```
void disable_irq (unsigned int irq);
```

### Arguments

*irq*

Interrupt to disable

## Description

Disable the selected interrupt line. Enables and Disables are nested. This function waits for any pending IRQ handlers for this interrupt to complete before returning. If you use this function while holding a resource the IRQ handler may need you will deadlock.

This function may be called - with care - from IRQ context.

## enable\_irq

### Name

`enable_irq` — enable handling of an irq

### Synopsis

```
void enable_irq (unsigned int irq);
```

### Arguments

*irq*

Interrupt to enable

## Description

Undoes the effect of one call to `disable_irq`. If this matches the last disable, processing of interrupts on this IRQ line is re-enabled.

This function may be called from IRQ context.

# probe\_irq\_mask

## Name

`probe_irq_mask` — scan a bitmap of interrupt lines

## Synopsis

```
unsigned int probe_irq_mask (unsigned long val);
```

## Arguments

*val*

mask of interrupts to consider

## Description

Scan the ISA bus interrupt lines and return a bitmap of active interrupts. The interrupt probe logic state is then returned to its previous value.

## Note

we need to scan all the irq's even though we will only return ISA irq numbers - just so that we reset them all to a known state.

## 10.2. MTRR Handling

### mtrr\_add

#### Name

`mtrr_add` — Add a memory type region

#### Synopsis

```
int mtrr_add (unsigned long base, unsigned long size, unsigned  
int type, char increment);
```

## Arguments

*base*

Physical base address of region

*size*

Physical size of region

*type*

Type of MTRR desired

*increment*

If this is true do usage counting on the region

## Description

Memory type region registers control the caching on newer Intel and non Intel processors. This function allows drivers to request an MTRR is added. The details and hardware specifics of each processor's implementation are hidden from the caller, but nevertheless the caller should expect to need to provide a power of two size on an equivalent power of two boundary.

If the region cannot be added either because all regions are in use or the CPU cannot support it a negative value is returned. On success the register number for this entry is returned, but should be treated as a cookie only.

On a multiprocessor machine the changes are made to all processors. This is required on x86 by the Intel processors.

The available types are

MTRR\_TYPE\_UNCACHABLE - No caching

MTRR\_TYPE\_WRBACK - Write data back in bursts whenever

MTRR\_TYPE\_WRCOMB - Write data back soon but allow bursts

MTRR\_TYPE\_WRTHROUGH - Cache reads but not writes

## BUGS

Needs a quiet flag for the cases where drivers do not mind failures and do not wish system log messages to be sent.

## mtrr\_del

### Name

`mtrr_del` — delete a memory type region

### Synopsis

```
int mtrr_del (int reg, unsigned long base, unsigned long size);
```

### Arguments

*reg*

Register returned by `mtrr_add`



*base*

Physical base address

*size*

Size of region

## Description

If register is supplied then base and size are ignored. This is how drivers should call it.

Releases an MTRR region. If the usage count drops to zero the register is freed and the region returns to default state. On success the register is returned, on failure a negative error code.

## 10.3. PCI Support Library

### **pci\_find\_slot**

#### **Name**

`pci_find_slot` — locate PCI device from a given PCI slot

## Synopsis

```
struct pci_dev * pci_find_slot (unsigned int bus, unsigned int  
devfn);
```

## Arguments

*bus*

number of PCI bus on which desired PCI device resides

*devfn*

encodes number of PCI slot in which the desired PCI device resides and the logical device number within that slot in case of multi-function devices.

## Description

Given a PCI bus and slot/function number, the desired PCI device is located in system global list of PCI devices. If the device is found, a pointer to its data structure is returned. If no device is found, `NULL` is returned.

## pci\_find\_subsys

### Name

`pci_find_subsys` — begin or continue searching for a PCI device by vendor/subvendor/device/subdevice id

### Synopsis

```
struct pci_dev * pci_find_subsys (unsigned int vendor, unsigned  
int device, unsigned int ss_vendor, unsigned int ss_device,  
const struct pci_dev * from);
```

### Arguments

*vendor*

PCI vendor id to match, or `PCI_ANY_ID` to match all vendor ids

*device*

PCI device id to match, or `PCI_ANY_ID` to match all device ids

*ss\_vendor*

PCI subsystem vendor id to match, or `PCI_ANY_ID` to match all vendor ids

*ss\_device*

PCI subsystem device id to match, or `PCI_ANY_ID` to match all device ids

*from*

Previous PCI device found in search, or NULL for new search.

## Description

Iterates through the list of known PCI devices. If a PCI device is found with a matching *vendor*, *device*, *ss\_vendor* and *ss\_device*, a pointer to its device structure is returned. Otherwise, NULL is returned. A new search is initiated by passing NULL to the *from* argument. Otherwise if *from* is not NULL, searches continue from next device on the global list.

## pci\_find\_device

### Name

`pci_find_device` — begin or continue searching for a PCI device by vendor/device id

### Synopsis

```
struct pci_dev * pci_find_device (unsigned int vendor, unsigned
int device, const struct pci_dev * from);
```

## Arguments

*vendor*

PCI vendor id to match, or `PCI_ANY_ID` to match all vendor ids

*device*

PCI device id to match, or `PCI_ANY_ID` to match all device ids

*from*

Previous PCI device found in search, or `NULL` for new search.

## Description

Iterates through the list of known PCI devices. If a PCI device is found with a matching *vendor* and *device*, a pointer to its device structure is returned. Otherwise, `NULL` is returned. A new search is initiated by passing `NULL` to the *from* argument. Otherwise if *from* is not `NULL`, searches continue from next device on the global list.

## pci\_find\_class

### Name

`pci_find_class` — begin or continue searching for a PCI device by class

## Synopsis

```
struct pci_dev * pci_find_class (unsigned int class, const  
struct pci_dev * from);
```

## Arguments

*class*

search for a PCI device with this class designation

*from*

Previous PCI device found in search, or NULL for new search.

## Description

Iterates through the list of known PCI devices. If a PCI device is found with a matching *class*, a pointer to its device structure is returned. Otherwise, NULL is returned. A new search is initiated by passing NULL to the *from* argument. Otherwise if *from* is not NULL, searches continue from next device on the global list.

## pci\_find\_capability

### Name

`pci_find_capability` — query for devices' capabilities

### Synopsis

```
int pci_find_capability (struct pci_dev * dev, int cap);
```

### Arguments

*dev*

PCI device to query

*cap*

capability code

### Description

Tell if a device supports a given PCI capability. Returns the address of the requested capability structure within the device's PCI configuration space or 0 in case the device does not support it. Possible values for *cap*:

`PCI_CAP_ID_PM` Power Management

`PCI_CAP_ID_AGP` Accelerated Graphics Port

PCI\_CAP\_ID\_VPD Vital Product Data  
PCI\_CAP\_ID\_SLOTID Slot Identification  
PCI\_CAP\_ID\_MSI Message Signalled Interrupts  
PCI\_CAP\_ID\_CHSWP CompactPCI HotSwap

## pci\_find\_parent\_resource

### Name

`pci_find_parent_resource` — return resource region of parent bus of given region

### Synopsis

```
struct resource * pci_find_parent_resource (const struct pci_dev  
* dev, struct resource * res);
```

### Arguments

*dev*

PCI device structure contains resources to be searched



*res*

child resource record for which parent is sought

## Description

For given resource region of given device, return the resource region of parent bus the given region is contained in or where it should be allocated from.

# pci\_set\_power\_state

## Name

`pci_set_power_state` — Set the power state of a PCI device

## Synopsis

```
int pci_set_power_state (struct pci_dev * dev, int state);
```

## Arguments

*dev*

PCI device to be suspended

*state*

Power state we're entering

## Description

Transition a device to a new power state, using the Power Management Capabilities in the device's config space.

## RETURN VALUE

-EINVAL if trying to enter a lower state than we're already in. 0 if we're already in the requested state. -EIO if device does not support PCI PM. 0 if we can successfully change the power state.

# pci\_save\_state

## Name

`pci_save_state` — save the PCI configuration space of a device before suspending

## Synopsis

```
int pci_save_state (struct pci_dev * dev, u32 * buffer);
```

## Arguments

*dev*

- PCI device that we're dealing with

*buffer*

- buffer to hold config space context

## Description

*buffer* must be large enough to hold the entire PCI 2.2 config space ( $\geq 64$  bytes).

# pci\_restore\_state

## Name

`pci_restore_state` — Restore the saved state of a PCI device

## Synopsis

```
int pci_restore_state (struct pci_dev * dev, u32 * buffer);
```

## Arguments

*dev*

- PCI device that we're dealing with

*buffer*

- saved PCI config space

## pci\_enable\_device

### Name

`pci_enable_device` — Initialize device before it's used by a driver.

### Synopsis

```
int pci_enable_device (struct pci_dev * dev);
```

## Arguments

*dev*

PCI device to be initialized

## Description

Initialize device before it's used by a driver. Ask low-level code to enable I/O and memory. Wake up the device if it was suspended. Beware, this function can fail.

## pci\_disable\_device

### Name

`pci_disable_device` — Disable PCI device after use

### Synopsis

```
void pci_disable_device (struct pci_dev * dev);
```

### Arguments

*dev*

PCI device to be disabled

## Description

Signal to the system that the PCI device is not in use by the system anymore. This only involves disabling PCI bus-mastering, if active.

# pci\_enable\_wake

## Name

`pci_enable_wake` — enable device to generate PME# when suspended

## Synopsis

```
int pci_enable_wake (struct pci_dev * dev, u32 state, int
enable);
```

## Arguments

*dev*

- PCI device to operate on

*state*

- Current state of device.

*enable*

- Flag to enable or disable generation

## Description

Set the bits in the device's PM Capabilities to generate PME# when the system is suspended.

-EIO is returned if device doesn't have PM Capabilities. -EINVAL is returned if device supports it, but can't generate wake events. 0 if operation is successful.

## pci\_release\_regions

### Name

`pci_release_regions` — Release reserved PCI I/O and memory resources

### Synopsis

```
void pci_release_regions (struct pci_dev * pdev);
```

## Arguments

*pdev*

PCI device whose resources were previously reserved by `pci_request_regions`

## Description

Releases all PCI I/O and memory resources previously reserved by a successful call to `pci_request_regions`. Call this function only after all use of the PCI regions has ceased.

# pci\_request\_regions

## Name

`pci_request_regions` — Reserved PCI I/O and memory resources

## Synopsis

```
int pci_request_regions (struct pci_dev * pdev, char *  
res_name);
```



## Arguments

*pdev*

PCI device whose resources are to be reserved

*res\_name*

Name to be associated with resource.

## Description

Mark all PCI regions associated with PCI device *pdev* as being reserved by owner *res\_name*. Do not access any address inside the PCI regions unless this call returns successfully.

Returns 0 on success, or `EBUSY` on error. A warning message is also printed on failure.

## pci\_match\_device

### Name

`pci_match_device` — Tell if a PCI device structure has a matching PCI device id structure

### Synopsis

```
const struct pci_device_id * pci_match_device (const struct
```

```
pci_device_id * ids, const struct pci_dev * dev);
```

## Arguments

*ids*

array of PCI device id structures to search in

*dev*

the PCI device structure to match against

## Description

Used by a driver to check whether a PCI device present in the system is in its list of supported devices. Returns the matching `pci_device_id` structure or `NULL` if there is no match.

# pci\_register\_driver

## Name

`pci_register_driver` — register a new pci driver

## Synopsis

```
int pci_register_driver (struct pci_driver * drv);
```

## Arguments

*drv*

the driver structure to register

## Description

Adds the driver structure to the list of registered drivers Returns the number of pci devices which were claimed by the driver during registration. The driver remains registered even if the return value is zero.

## pci\_unregister\_driver

### Name

`pci_unregister_driver` — unregister a pci driver

## Synopsis

```
void pci_unregister_driver (struct pci_driver * drv);
```

## Arguments

*drv*

the driver structure to unregister

## Description

Deletes the driver structure from the list of registered PCI drivers, gives it a chance to clean up by calling its `remove` function for each device it was responsible for, and marks those devices as driverless.

## pci\_announce\_device\_to\_drivers

### Name

`pci_announce_device_to_drivers` — tell the drivers a new device has appeared

## Synopsis

```
void pci_announce_device_to_drivers (struct pci_dev * dev);
```

## Arguments

*dev*

the device that has shown up

## Description

Notifys the drivers that a new device has appeared, and also notifys userspace through /sbin/hotplug.

# pci\_insert\_device

## Name

pci\_insert\_device — insert a hotplug device

## Synopsis

```
void pci_insert_device (struct pci_dev * dev, struct pci_bus *  
bus);
```

## Arguments

*dev*

the device to insert

*bus*

where to insert it

## Description

Add a new device to the device lists and notify userspace (/sbin/hotplug).

## pci\_remove\_device

### Name

`pci_remove_device` — remove a hotplug device

## Synopsis

```
void pci_remove_device (struct pci_dev * dev);
```

## Arguments

*dev*

the device to remove

## Description

Delete the device structure from the device lists and notify userspace (/sbin/hotplug).

## pci\_dev\_driver

### Name

`pci_dev_driver` — get the `pci_driver` of a device

## Synopsis

```
struct pci_driver * pci_dev_driver (const struct pci_dev * dev);
```

## Arguments

*dev*

the device to query

## Description

Returns the appropriate `pci_driver` structure or `NULL` if there is no registered driver for the device.

# pci\_set\_master

## Name

`pci_set_master` — enables bus-mastering for device `dev`

## Synopsis

```
void pci_set_master (struct pci_dev * dev);
```



## Arguments

*dev*

the PCI device to enable

## Description

Enables bus-mastering on the device and calls `pcibios_set_master` to do the needed arch specific settings.

# pci\_setup\_device

## Name

`pci_setup_device` — fill in class and map information of a device

## Synopsis

```
int pci_setup_device (struct pci_dev * dev);
```

## Arguments

*dev*

the device structure to fill

## Description

Initialize the device structure with information about the device's vendor, class, memory and IO-space addresses, IRQ lines etc. Called at initialisation of the PCI subsystem and by CardBus services. Returns 0 on success and -1 if unknown type of device (not normal, bridge or CardBus).

## pci\_pool\_create

### Name

`pci_pool_create` — Creates a pool of pci consistent memory blocks, for dma.

### Synopsis

```
struct pci_pool * pci_pool_create (const char * name, struct  
pci_dev * pdev, size_t size, size_t align, size_t allocation,  
int flags);
```

## Arguments

*name*

name of pool, for diagnostics

*pdev*

pci device that will be doing the DMA

*size*

size of the blocks in this pool.

*align*

alignment requirement for blocks; must be a power of two

*allocation*

returned blocks won't cross this boundary (or zero)

*flags*

SLAB\_\* flags (not all are supported).

## Description

Returns a pci allocation pool with the requested characteristics, or null if one can't be created. Given one of these pools, `pci_pool_alloc` may be used to allocate memory. Such memory will all have “consistent” DMA mappings, accessible by the device and its driver without using cache flushing primitives. The actual size of blocks allocated may be larger than requested because of alignment.

If allocation is nonzero, objects returned from `pci_pool_alloc` won't cross that size boundary. This is useful for devices which have addressing restrictions on individual DMA transfers, such as not crossing boundaries of 4KBytes.

## pci\_pool\_destroy

### Name

`pci_pool_destroy` — destroys a pool of pci memory blocks.

### Synopsis

```
void pci_pool_destroy (struct pci_pool * pool);
```

### Arguments

*pool*

pci pool that will be destroyed

### Description

Caller guarantees that no more memory from the pool is in use, and that nothing will try to use the pool after this call.

## pci\_pool\_alloc

### Name

`pci_pool_alloc` — get a block of consistent memory

### Synopsis

```
void * pci_pool_alloc (struct pci_pool * pool, int mem_flags,  
dma_addr_t * handle);
```

### Arguments

*pool*

pci pool that will produce the block

*mem\_flags*

SLAB\_KERNEL or SLAB\_ATOMIC

*handle*

pointer to dma address of block

### Description

This returns the kernel virtual address of a currently unused block, and reports its dma address through the handle. If such a memory block can't be allocated, null is returned.

## pci\_pool\_free

### Name

`pci_pool_free` — put block back into pci pool

### Synopsis

```
void pci_pool_free (struct pci_pool * pool, void * vaddr,  
dma_addr_t dma);
```

### Arguments

*pool*

the pci pool holding the block

*vaddr*

virtual address of block

*dma*

dma address of block

## **Description**

Caller promises neither device nor driver will again touch this block unless it is first re-allocated.

## **10.4. MCA Architecture**

### **10.4.1. MCA Device Functions**

#### **mca\_find\_adapter**

##### **Name**

`mca_find_adapter` — scan for adapters

##### **Synopsis**

```
int mca_find_adapter (int id, int start);
```

## Arguments

*id*

MCA identification to search for

*start*

starting slot

## Description

Search the MCA configuration for adapters matching the 16bit ID given. The first time it should be called with start as zero and then further calls made passing the return value of the previous call until MCA\_NOTFOUND is returned.

Disabled adapters are not reported.

## mca\_find\_unused\_adapter

### Name

`mca_find_unused_adapter` — scan for unused adapters

### Synopsis

```
int mca_find_unused_adapter (int id, int start);
```



## Arguments

*id*

MCA identification to search for

*start*

starting slot

## Description

Search the MCA configuration for adapters matching the 16bit ID given. The first time it should be called with start as zero and then further calls made passing the return value of the previous call until MCA\_NOTFOUND is returned.

Adapters that have been claimed by drivers and those that are disabled are not reported. This function thus allows a driver to scan for further cards when some may already be driven.

## mca\_read\_stored\_pos

### Name

mca\_read\_stored\_pos — read POS register from boot data

## Synopsis

```
unsigned char mca_read_stored_pos (int slot, int reg);
```

## Arguments

*slot*

slot number to read from

*reg*

register to read from

## Description

Fetch a POS value that was stored at boot time by the kernel when it scanned the MCA space. The register value is returned. Missing or invalid registers report 0.

## mca\_read\_pos

### Name

mca\_read\_pos — read POS register from card

## Synopsis

```
unsigned char mca_read_pos (int slot, int reg);
```

## Arguments

*slot*

slot number to read from

*reg*

register to read from

## Description

Fetch a POS value directly from the hardware to obtain the current value. This is much slower than `mca_read_stored_pos` and may not be invoked from interrupt context. It handles the deep magic required for onboard devices transparently.

## **mca\_write\_pos**

### Name

`mca_write_pos` — read POS register from card

## Synopsis

```
void mca_write_pos (int slot, int reg, unsigned char byte);
```

## Arguments

*slot*

slot number to read from

*reg*

register to read from

*byte*

byte to write to the POS registers

## Description

Store a POS value directly from the hardware. You should not normally need to use this function and should have a very good knowledge of MCA bus before you do so. Doing this wrongly can damage the hardware.

This function may not be used from interrupt context.

Note that this is technically a Bad Thing, as IBM tech stuff says you should only set POS values through their utilities. However, some devices such as the 3c523 recommend that you write back some data to make sure the configuration is consistent. I'd say that IBM is right, but I like my drivers to work.

This function can't do checks to see if multiple devices end up with the same resources, so you might see magic smoke if someone screws up.

## **mca\_set\_adapter\_name**

### **Name**

`mca_set_adapter_name` — Set the description of the card

### **Synopsis**

```
void mca_set_adapter_name (int slot, char* name);
```

### **Arguments**

*slot*

slot to name

*name*

text string for the namen

### **Description**

This function sets the name reported via `/proc` for this adapter slot. This is for user information only. Setting a name deletes any previous name.

## **mca\_set\_adapter\_procfn**

### **Name**

`mca_set_adapter_procfn` — Set the /proc callback

### **Synopsis**

```
void mca_set_adapter_procfn (int slot, MCA_ProcFn procfn, void*  
dev);
```

### **Arguments**

*slot*

slot to configure

*procfn*

callback function to call for /proc

*dev*

device information passed to the callback

## Description

This sets up an information callback for `/proc/mca/slot?`. The function is called with the buffer, slot, and device pointer (or some equally informative context information, or nothing, if you prefer), and is expected to put useful information into the buffer. The adapter name, ID, and POS registers get printed before this is called though, so don't do it again.

This should be called with a `NULL procfn` when a module unregisters, thus preventing kernel crashes and other such nastiness.

## `mca_is_adapter_used`

### Name

`mca_is_adapter_used` — check if claimed by driver

### Synopsis

```
int mca_is_adapter_used (int slot);
```

## Arguments

*slot*

slot to check

## Description

Returns 1 if the slot has been claimed by a driver

# mca\_mark\_as\_used

## Name

`mca_mark_as_used` — claim an MCA device

## Synopsis

```
int mca_mark_as_used (int slot);
```



## Arguments

*slot*

slot to claim

## FIXME

should we make this threadsafe

Claim an MCA slot for a device driver. If the slot is already taken the function returns 1, if it is not taken it is claimed and 0 is returned.

## mca\_mark\_as\_unused

### Name

`mca_mark_as_unused` — release an MCA device

### Synopsis

```
void mca_mark_as_unused (int slot);
```

## Arguments

*slot*

slot to claim

## Description

Release the slot for other drives to use.

# mca\_get\_adapter\_name

## Name

`mca_get_adapter_name` — get the adapter description

## Synopsis

```
char * mca_get_adapter_name (int slot);
```

## Arguments

*slot*

slot to query

## Description

Return the adapter description if set. If it has not been set or the slot is out range then return NULL.

# mca\_isadapter

## Name

`mca_isadapter` — check if the slot holds an adapter

## Synopsis

```
int mca_isadapter (int slot);
```

## Arguments

*slot*

slot to query

## Description

Returns zero if the slot does not hold an adapter, non zero if it does.

# mca\_isenabled

## Name

`mca_isenabled` — check if the slot holds an adapter

## Synopsis

```
int mca_isenabled (int slot);
```

## Arguments

*slot*

slot to query

## Description

Returns a non zero value if the slot holds an enabled adapter and zero for any other case.

### 10.4.2. MCA Bus DMA

## **mca\_enable\_dma**

### Name

`mca_enable_dma` — channel to enable DMA on

### Synopsis

```
void mca_enable_dma (unsigned int dmanr);
```

## Arguments

*dmanr*

DMA channel

## Description

Enable the MCA bus DMA on a channel. This can be called from IRQ context.

# mca\_disable\_dma

## Name

`mca_disable_dma` — channel to disable DMA on

## Synopsis

```
void mca_disable_dma (unsigned int dmanr);
```

## Arguments

*dmanr*

DMA channel

## Description

Enable the MCA bus DMA on a channel. This can be called from IRQ context.

# mca\_set\_dma\_addr

## Name

`mca_set_dma_addr` — load a 24bit DMA address

## Synopsis

```
void mca_set_dma_addr (unsigned int dmanr, unsigned int a);
```

## Arguments

*dmanr*

DMA channel

*a*

24bit bus address

## Description

Load the address register in the DMA controller. This has a 24bit limitation (16Mb).

## **mca\_get\_dma\_addr**

### Name

`mca_get_dma_addr` — load a 24bit DMA address

## Synopsis

```
unsigned int mca_get_dma_addr (unsigned int dmanr);
```



## Arguments

*dmanr*

DMA channel

## Description

Read the address register in the DMA controller. This has a 24bit limitation (16Mb).  
The return is a bus address.

# mca\_set\_dma\_count

## Name

`mca_set_dma_count` — load a 16bit transfer count

## Synopsis

```
void mca_set_dma_count (unsigned int dmanr, unsigned int count);
```

## Arguments

*dmanr*

DMA channel

*count*

count

## Description

Set the DMA count for this channel. This can be up to 64Kbytes. Setting a count of zero will not do what you expect.

# mca\_get\_dma\_residue

## Name

`mca_get_dma_residue` — get the remaining bytes to transfer

## Synopsis

```
unsigned int mca_get_dma_residue (unsigned int dmanr);
```

## Arguments

*dmanr*

DMA channel

## Description

This function returns the number of bytes left to transfer on this DMA channel.

# mca\_set\_dma\_io

## Name

`mca_set_dma_io` — set the port for an I/O transfer

## Synopsis

```
void mca_set_dma_io (unsigned int dmanr, unsigned int io_addr);
```

## Arguments

*dmanr*

DMA channel

*io\_addr*

an I/O port number

## Description

Unlike the ISA bus DMA controllers the DMA on MCA bus can transfer with an I/O port target.

# mca\_set\_dma\_mode

## Name

`mca_set_dma_mode` — set the DMA mode

## Synopsis

```
void mca_set_dma_mode (unsigned int dmanr, unsigned int mode);
```

## Arguments

*dmanr*

DMA channel

*mode*

mode to set

## Description

The DMA controller supports several modes. The mode values you can

### set are

`MCA_DMA_MODE_READ` when reading from the DMA device.

`MCA_DMA_MODE_WRITE` to writing to the DMA device.

`MCA_DMA_MODE_IO` to do DMA to or from an I/O port.

`MCA_DMA_MODE_16` to do 16bit transfers.

# Chapter 11. The Device File System

## devfs\_put

### Name

`devfs_put` — Put (release) a reference to a devfs entry.

### Synopsis

```
void devfs_put (devfs_handle_t de);
```

### Arguments

*de*

The handle to the devfs entry.

# devfs\_register

## Name

`devfs_register` — Register a device entry.

## Synopsis

```
devfs_handle_t devfs_register (devfs_handle_t dir, const char *  
name, unsigned int flags, unsigned int major, unsigned int  
minor, umode_t mode, void * ops, void * info);
```

## Arguments

*dir*

The handle to the parent devfs directory entry. If this is `NULL` the new name is relative to the root of the devfs.

*name*

The name of the entry.

*flags*

A set of bitwise-ORed flags (`DEVFS_FL_*`).

*major*

The major number. Not needed for regular files.

*minor*

The minor number. Not needed for regular files.

*mode*

The default file mode.

*ops*

The `&file_operations` or `&block_device_operations` structure. This must not be externally deallocated.

*info*

An arbitrary pointer which will be written to the *private\_data* field of the `&file` structure passed to the device driver. You can set this to whatever you like, and change it once the file is opened (the next file opened will not see this change).

## Description

Returns a handle which may later be used in a call to `devfs_unregister`. On failure `NULL` is returned.

# devfs\_unregister

## Name

`devfs_unregister` — Unregister a device entry.



## Synopsis

```
void devfs_unregister (devfs_handle_t de);
```

## Arguments

*de*

A handle previously created by `devfs_register` or returned from `devfs_get_handle`. If this is `NULL` the routine does nothing.

## **devfs\_mk\_symlink**

### Name

`devfs_mk_symlink` —

## Synopsis

```
int devfs_mk_symlink (devfs_handle_t dir, const char * name,  
unsigned int flags, const char * link, devfs_handle_t * handle,  
void * info);
```

## Arguments

*dir*

The handle to the parent devfs directory entry. If this is `NULL` the new name is relative to the root of the devfs.

*name*

The name of the entry.

*flags*

A set of bitwise-ORed flags (`DEVFS_FL_*`).

*link*

The destination name.

*handle*

The handle to the symlink entry is written here. This may be `NULL`.

*info*

An arbitrary pointer which will be associated with the entry.

## Description

Returns 0 on success, else a negative error code is returned.

## devfs\_mk\_dir

### Name

`devfs_mk_dir` — Create a directory in the devfs namespace.

### Synopsis

```
devfs_handle_t devfs_mk_dir (devfs_handle_t dir, const char *  
name, void * info);
```

### Arguments

*dir*

The handle to the parent devfs directory entry. If this is `NULL` the new name is relative to the root of the devfs.

*name*

The name of the entry.

*info*

An arbitrary pointer which will be associated with the entry.

## Description

Use of this function is optional. The `devfs_register` function will automatically create intermediate directories as needed. This function is provided for efficiency reasons, as it provides a handle to a directory. Returns a handle which may later be used in a call to `devfs_unregister`. On failure `NULL` is returned.

## `devfs_get_handle`

### Name

`devfs_get_handle` — Find the handle of a `devfs` entry.

### Synopsis

```
devfs_handle_t devfs_get_handle (devfs_handle_t dir, const char  
* name, unsigned int major, unsigned int minor, char type, int  
traverse_symlinks);
```

### Arguments

*dir*

The handle to the parent `devfs` directory entry. If this is `NULL` the name is relative to the root of the `devfs`.

*name*

The name of the entry.

*major*

The major number. This is used if *name* is NULL.

*minor*

The minor number. This is used if *name* is NULL.

*type*

The type of special file to search for. This may be either DEVFS\_SPECIAL\_CHR or DEVFS\_SPECIAL\_BLK.

*traverse\_symlinks*

If TRUE then symlink entries in the devfs namespace are traversed. Symlinks pointing out of the devfs namespace will cause a failure. Symlink traversal consumes stack space.

## Description

Returns a handle which may later be used in a call to `devfs_unregister`, `devfs_get_flags`, or `devfs_set_flags`. A subsequent `devfs_put` is required to decrement the refcount. On failure NULL is returned.

## devfs\_get\_flags

### Name

`devfs_get_flags` — Get the flags for a devfs entry.

### Synopsis

```
int devfs_get_flags (devfs_handle_t de, unsigned int * flags);
```

### Arguments

*de*

The handle to the device entry.

*flags*

The flags are written here.

### Description

Returns 0 on success, else a negative error code.

## devfs\_get\_maj\_min

### Name

`devfs_get_maj_min` — Get the major and minor numbers for a devfs entry.

### Synopsis

```
int devfs_get_maj_min (devfs_handle_t de, unsigned int * major,  
unsigned int * minor);
```

### Arguments

*de*

The handle to the device entry.

*major*

The major number is written here. This may be NULL.

*minor*

The minor number is written here. This may be NULL.

### Description

Returns 0 on success, else a negative error code.

## devfs\_get\_handle\_from\_inode

### Name

`devfs_get_handle_from_inode` — Get the devfs handle for a VFS inode.

### Synopsis

```
devfs_handle_t devfs_get_handle_from_inode (struct inode *  
inode);
```

### Arguments

*inode*

The VFS inode.

### Description

Returns the devfs handle on success, else NULL.



## devfs\_generate\_path

### Name

`devfs_generate_path` — Generate a pathname for an entry, relative to the devfs root.

### Synopsis

```
int devfs_generate_path (devfs_handle_t de, char * path, int
buflen);
```

### Arguments

*de*

The devfs entry.

*path*

The buffer to write the pathname to. The pathname and '\0' terminator will be written at the end of the buffer.

*buflen*

The length of the buffer.

## Description

Returns the offset in the buffer where the pathname starts on success, else a negative error code.

# devfs\_get\_ops

## Name

`devfs_get_ops` — Get the device operations for a devfs entry.

## Synopsis

```
void * devfs_get_ops (devfs_handle_t de);
```

## Arguments

*de*

The handle to the device entry.

## Description

Returns a pointer to the device operations on success, else NULL. The use count for the module owning the operations will be incremented.

# devfs\_set\_file\_size

## Name

`devfs_set_file_size` — Set the file size for a devfs regular file.

## Synopsis

```
int devfs_set_file_size (devfs_handle_t de, unsigned long size);
```

## Arguments

*de*

The handle to the device entry.

*size*

The new file size.

## Description

Returns 0 on success, else a negative error code.

# devfs\_get\_info

## Name

`devfs_get_info` — Get the info pointer written to `private_data` of `de` upon open.

## Synopsis

```
void * devfs_get_info (devfs_handle_t de);
```

## Arguments

*de*

The handle to the device entry.

## Description

Returns the info pointer.

## devfs\_set\_info

### Name

`devfs_set_info` — Set the info pointer written to `private_data` upon open.

### Synopsis

```
int devfs_set_info (devfs_handle_t de, void * info);
```

### Arguments

*de*

The handle to the device entry.

*info*

pointer to the data

### Description

Returns 0 on success, else a negative error code.

## devfs\_get\_parent

### Name

`devfs_get_parent` — Get the parent device entry.

### Synopsis

```
devfs_handle_t devfs_get_parent (devfs_handle_t de);
```

### Arguments

*de*

The handle to the device entry.

### Description

Returns the parent device entry if it exists, else `NULL`.

## **devfs\_get\_first\_child**

### **Name**

`devfs_get_first_child` — Get the first leaf node in a directory.

### **Synopsis**

```
devfs_handle_t devfs_get_first_child (devfs_handle_t de);
```

### **Arguments**

*de*

The handle to the device entry.

### **Description**

Returns the leaf node device entry if it exists, else `NULL`.

## devfs\_get\_next\_sibling

### Name

`devfs_get_next_sibling` — Get the next sibling leaf node. for a device entry.

### Synopsis

```
devfs_handle_t devfs_get_next_sibling (devfs_handle_t de);
```

### Arguments

*de*

The handle to the device entry.

### Description

Returns the leaf node device entry if it exists, else `NULL`.



## devfs\_auto\_unregister

### Name

`devfs_auto_unregister` — Configure a devfs entry to be automatically unregistered.

### Synopsis

```
void devfs_auto_unregister (devfs_handle_t master,  
devfs_handle_t slave);
```

### Arguments

*master*

The master devfs entry. Only one slave may be registered.

*slave*

The devfs entry which will be automatically unregistered when the master entry is unregistered. It is illegal to call `devfs_unregister` on this entry.

## devfs\_get\_unregister\_slave

### Name

`devfs_get_unregister_slave` — Get the slave entry which will be automatically unregistered.

### Synopsis

```
devfs_handle_t devfs_get_unregister_slave (devfs_handle_t
master);
```

### Arguments

*master*

The master devfs entry.

### Description

Returns the slave which will be unregistered when *master* is unregistered.

## devfs\_get\_name

### Name

`devfs_get_name` — Get the name for a device entry in its parent directory.

### Synopsis

```
const char * devfs_get_name (devfs_handle_t de, unsigned int *  
namelen);
```

### Arguments

*de*

The handle to the device entry.

*namelen*

The length of the name is written here. This may be `NULL`.

### Description

Returns the name on success, else `NULL`.

## devfs\_register\_chrdev

### Name

`devfs_register_chrdev` — Optionally register a conventional character driver.

### Synopsis

```
int devfs_register_chrdev (unsigned int major, const char *  
name, struct file_operations * fops);
```

### Arguments

*major*

The major number for the driver.

*name*

The name of the driver (as seen in `/proc/devices`).

*fops*

The `&file_operations` structure pointer.

### Description

This function will register a character driver provided the “`devfs=only`” option was not provided at boot time. Returns 0 on success, else a negative error code on failure.

## devfs\_register\_blkdev

### Name

`devfs_register_blkdev` — Optionally register a conventional block driver.

### Synopsis

```
int devfs_register_blkdev (unsigned int major, const char *  
name, struct block_device_operations * bdops);
```

### Arguments

*major*

The major number for the driver.

*name*

The name of the driver (as seen in `/proc/devices`).

*bdops*

The `&block_device_operations` structure pointer.

## Description

This function will register a block driver provided the “devfs=only” option was not provided at boot time. Returns 0 on success, else a negative error code on failure.

# devfs\_unregister\_chrdev

## Name

`devfs_unregister_chrdev` — Optionally unregister a conventional character driver.

## Synopsis

```
int devfs_unregister_chrdev (unsigned int major, const char *  
name);
```

## Arguments

*major*

The major number for the driver.

*name*

The name of the driver (as seen in `/proc/devices`).

## Description

This function will unregister a character driver provided the “devfs=only” option was not provided at boot time. Returns 0 on success, else a negative error code on failure.

# devfs\_unregister\_blkdev

## Name

`devfs_unregister_blkdev` — Optionally unregister a conventional block driver.

## Synopsis

```
int devfs_unregister_blkdev (unsigned int major, const char *  
name);
```

## Arguments

*major*

The major number for the driver.

*name*

The name of the driver (as seen in `/proc/devices`).

## **Description**

This function will unregister a block driver provided the “devfs=only” option was not provided at boot time. Returns 0 on success, else a negative error code on failure.



# Chapter 12. Power Management

## pm\_register

### Name

`pm_register` — register a device with power management

### Synopsis

```
struct pm_dev * pm_register (pm_dev_t type, unsigned long id,  
pm_callback callback);
```

### Arguments

*type*

device type

*id*

device ID

*callback*

callback function

## Description

Add a device to the list of devices that wish to be notified about power management events. A `&pm_dev` structure is returned on success, on failure the return is `NULL`.

The callback function will be called in process context and it may sleep.

## pm\_unregister

### Name

`pm_unregister` — unregister a device with power management

### Synopsis

```
void pm_unregister (struct pm_dev * dev);
```

### Arguments

*dev*

device to unregister

## Description

Remove a device from the power management notification lists. The dev passed must be a handle previously returned by pm\_register.

# pm\_unregister\_all

## Name

pm\_unregister\_all — unregister all devices with matching callback

## Synopsis

```
void pm_unregister_all (pm_callback callback);
```

## Arguments

*callback*

callback function pointer

## Description

Unregister every device that would call the callback passed. This is primarily meant as a helper function for loadable modules. It enables a module to give up all its managed devices without keeping its own private list.

## pm\_send

### Name

`pm_send` — send request to a single device

### Synopsis

```
int pm_send (struct pm_dev * dev, pm_request_t rqst, void *  
data);
```

### Arguments

*dev*

device to send to

*rqst*

power management request

*data*

data for the callback

## Description

Issue a power management request to a given device. The `PM_SUSPEND` and `PM_RESUME` events are handled specially. The data field must hold the intended next state. No call is made if the state matches.

## BUGS

what stops two power management requests occurring in parallel and conflicting.

## WARNING

Calling `pm_send` directly is not generally recommended, in particular there is no locking against the `pm_dev` going away. The caller must maintain all needed locking or have 'inside knowledge' on the safety. Also remember that this function is not locked against `pm_unregister`. This means that you must handle SMP races on callback execution and unload yourself.

## pm\_send\_all

### Name

`pm_send_all` — send request to all managed devices

### Synopsis

```
int pm_send_all (pm_request_t rqst, void * data);
```

### Arguments

*rqst*

power management request

*data*

data for the callback

### Description

Issue a power management request to a all devices. The `PM_SUSPEND` events are handled specially. Any device is permitted to fail a suspend by returning a non zero (error) value from its callback function. If any device vetoes a suspend request then all other devices that have suspended during the processing of this request are restored to their previous state.

## WARNING

This function takes the `pm_devs_lock`. The lock is not dropped until the callbacks have completed. This prevents races against pm locking functions, races against module unload `pm_unregister` code. It does mean however that you must not issue `pm_` functions within the callback or you will deadlock and users will hate you.

Zero is returned on success. If a suspend fails then the status from the device that vetoes the suspend is returned.

## BUGS

what stops two power management requests occurring in parallel and conflicting.

## pm\_find

### Name

`pm_find` — find a device

### Synopsis

```
struct pm_dev * pm_find (pm_dev_t type, struct pm_dev * from);
```

## Arguments

*type*

type of device

*from*

where to start looking

## Description

Scan the power management list for devices of a specific type. The return value for a matching device may be passed to further calls to this function to find further matches. A `NULL` indicates the end of the list.

To search from the beginning pass `NULL` as the *from* value.

The caller **MUST** hold the `pm_devs_lock` lock when calling this function. The instant that the lock is dropped all pointers returned may become invalid.



# Chapter 13. Block Devices

## blk\_cleanup\_queue

### Name

`blk_cleanup_queue` — release a `request_queue_t` when it is no longer needed

### Synopsis

```
void blk_cleanup_queue (request_queue_t * q);
```

### Arguments

*q*

the request queue to be released

### Description

`blk_cleanup_queue` is the pair to `blk_init_queue`. It should be called when a request queue is being released; typically when a block device is being de-registered. Currently, its primary task is to free all the `&struct request` structures that were allocated to the queue.

## Caveat

Hopefully the low level driver will have finished any outstanding requests first...

# blk\_queue\_headactive

## Name

`blk_queue_headactive` — indicate whether head of request queue may be active

## Synopsis

```
void blk_queue_headactive (request_queue_t * q, int active);
```

## Arguments

*q*

The queue which this applies to.

*active*

A flag indication where the head of the queue is active.

## Description

The driver for a block device may choose to leave the currently active request on the request queue, removing it only when it has completed. The queue handling routines assume this by default for safety reasons and will not involve the head of the request queue in any merging or reordering of requests when the queue is unplugged (and thus may be working on this particular request).

If a driver removes requests from the queue before processing them, then it may indicate that it does so, thereby allowing the head of the queue to be involved in merging and reordering. This is done by calling `blk_queue_headactive` with an *active* flag of 0.

If a driver processes several requests at once, it must remove them (or at least all but one of them) from the request queue.

When a queue is plugged the head will be assumed to be inactive.

## blk\_queue\_make\_request

### Name

`blk_queue_make_request` — define an alternate `make_request` function for a device

### Synopsis

```
void blk_queue_make_request (request_queue_t * q,
make_request_fn * mfn);
```

## Arguments

*q*

the request queue for the device to be affected

*mf n*

the alternate make\_request function

## Description

The normal way for `&struct buffer_heads` to be passed to a device driver is for them to be collected into requests on a request queue, and then to allow the device driver to select requests off that queue when it is ready. This works well for many block devices. However some block devices (typically virtual devices such as md or lvm) do not benefit from the processing on the request queue, and are served best by having the requests passed directly to them. This can be achieved by providing a function to `blk_queue_make_request`.

## Caveat

The driver that does this *must* be able to deal appropriately with buffers in “highmemory”, either by calling `bh_kmap` to get a kernel mapping, to by calling `create_bounce` to create a buffer in normal memory.

# blk\_init\_queue

## Name

`blk_init_queue` — prepare a request queue for use with a block device

## Synopsis

```
void blk_init_queue (request_queue_t * q, request_fn_proc *  
rfn);
```

## Arguments

*q*

The `&request_queue_t` to be initialised

*rfn*

The function to be called to process requests that have been placed on the queue.

## Description

If a block device wishes to use the standard request handling procedures, which sorts requests and coalesces adjacent requests, then it must call `blk_init_queue`. The function *rfn* will be called when there are requests on the queue that need to be processed. If the device supports plugging, then *rfn* may not be called immediately when requests are available on the queue, but may be called at some time later instead.

Plugged queues are generally unplugged when a buffer belonging to one of the requests on the queue is needed, or due to memory pressure.

*rfn* is not required, or even expected, to remove all requests off the queue, but only as many as it can handle at a time. If it does leave requests on the queue, it is responsible for arranging that the requests get dealt with eventually.

A global spin lock `$io_request_lock` must be held while manipulating the requests on the request queue.

The request on the head of the queue is by default assumed to be potentially active, and it is not considered for re-ordering or merging whenever the given queue is unplugged. This behaviour can be changed with `blk_queue_headactive`.

## Note

`blk_init_queue` must be paired with a `blk_cleanup_queue` call when the block device is deactivated (such as at module unload).

# generic\_make\_request

## Name

`generic_make_request` —

## Synopsis

```
void generic_make_request (int rw, struct buffer_head * bh);
```

## Arguments

*rw*

READ, WRITE, or READA - what sort of I/O is desired.

*bh*

The buffer head describing the location in memory and on the device.

## Description

`generic_make_request` is used to make I/O requests of block devices. It is passed a `&struct buffer_head` and a `&rw` value. The `READ` and `WRITE` options are (hopefully) obvious in meaning. The `READA` value means that a read is required, but that the driver is free to fail the request if, for example, it cannot get needed resources immediately.

`generic_make_request` does not return any status. The success/failure status of the request, along with notification of completion, is delivered asynchronously through the `bh->b_end_io` function described (one day) else where.

The caller of `generic_make_request` must make sure that `b_page`, `b_addr`, `b_size` are set to describe the memory buffer, that `b_rdev` and `b_rsector` are set to describe the device address, and the `b_end_io` and optionally `b_private` are set to describe how completion notification should be signaled. `BH_Mapped` should also be set (to confirm that `b_dev` and `b_blocknr` are valid).

`generic_make_request` and the drivers it calls may use `b_reqnext`, and may change `b_rdev` and `b_rsector`. So the values of these fields should NOT be depended on after the call to `generic_make_request`. Because of this, the caller should record the device address information in `b_dev` and `b_blocknr`.

Apart from those fields mentioned above, no other fields, and in particular, no other flags, are changed by `generic_make_request` or any lower level drivers.

## submit\_bh

### Name

`submit_bh` —

### Synopsis

```
void submit_bh (int rw, struct buffer_head * bh);
```

### Arguments

*rw*

whether to READ or WRITE, or maybe to READA (read ahead)

*bh*

The `&struct buffer_head` which describes the I/O



## Description

`submit_bh` is very similar in purpose to `generic_make_request`, and uses that function to do most of the work.

The extra functionality provided by `submit_bh` is to determine `b_rsector` from `b_blocknr` and `b_size`, and to set `b_rdev` from `b_dev`. This is appropriate for IO requests that come from the buffer cache and page cache which (currently) always use aligned blocks.

## ll\_rw\_block

### Name

`ll_rw_block` — level access to block devices

### Synopsis

```
void ll_rw_block (int rw, int nr, struct buffer_head * * bhs);
```

### Arguments

*rw*

whether to READ or WRITE or maybe READA (readahead)

*nr*number of `&struct buffer_heads` in the array*bhs*array of pointers to `&struct buffer_head`

## Description

`ll_rw_block` takes an array of pointers to `&struct buffer_heads`, and requests an I/O operation on them, either a `READ` or a `WRITE`. The third `READA` option is described in the documentation for `generic_make_request` which `ll_rw_block` calls.

This function provides extra functionality that is not in `generic_make_request` that is relevant to buffers in the buffer cache or page cache. In particular it drops any buffer that it cannot get a lock on (with the `BH_Lock` state bit), any buffer that appears to be clean when doing a write request, and any buffer that appears to be up-to-date when doing read request. Further it marks as clean buffers that are processed for writing (the buffer cache won't assume that they are actually clean until the buffer gets unlocked).

`ll_rw_block` sets `b_end_io` to simple completion handler that marks the buffer up-to-date (if appropriate), unlocks the buffer and wakes any waiters. As client that needs a more interesting completion routine should call `submit_bh` (or `generic_make_request`) directly.

## Caveat

All of the buffers must be for the same device, and must also be

# end\_that\_request\_first

## Name

`end_that_request_first` — end I/O on one buffer.

## Synopsis

```
int end_that_request_first (struct request * req, int uptodate,  
char * name);
```

## Arguments

*req*

the request being processed

*uptodate*

0 for I/O error

*name*

the name printed for an I/O error

## Description

Ends I/O on the first buffer attached to *req*, and sets it up for the next `buffer_head` (if any) in the cluster.

## **Return**

0 - we are done with this request, call `end_that_request_last` 1 - still buffers pending for this request

## **Caveat**

Drivers implementing their own `end_request` handling must call `blk_finished_io` appropriately.

# Chapter 14. Miscellaneous Devices

## misc\_register

### Name

`misc_register` — register a miscellaneous device

### Synopsis

```
int misc_register (struct miscdevice * misc);
```

### Arguments

*misc*

device structure

### Description

Register a miscellaneous device with the kernel. If the minor number is set to `MISC_DYNAMIC_MINOR` a minor number is assigned and placed in the minor field of the structure. For other cases the minor number requested is used.

The structure passed is linked into the kernel and may not be destroyed until it has been unregistered.

A zero is returned on success and a negative errno code for failure.

## **misc\_deregister**

### **Name**

`misc_deregister` — unregister a miscellaneous device

### **Synopsis**

```
int misc_deregister (struct miscdevice * misc);
```

### **Arguments**

*misc*

device to unregister

## **Description**

Unregister a miscellaneous device that was previously successfully registered with `misc_register`. Success is indicated by a zero return, a negative `errno` code indicates an error.

# Chapter 15. Video4Linux

## video\_unregister\_device

### Name

`video_unregister_device` — unregister a video4linux device

### Synopsis

```
void video_unregister_device (struct video_device * vfd);
```

### Arguments

*vfd*

the device to unregister

### Description

This unregisters the passed device and deassigns the minor number. Future open calls will be met with errors.



# Chapter 16. Sound Devices

## register\_sound\_special

### Name

`register_sound_special` — register a special sound node

### Synopsis

```
int register_sound_special (struct file_operations * fops, int  
unit);
```

### Arguments

*fops*

File operations for the driver

*unit*

Unit number to allocate

## Description

Allocate a special sound device by minor number from the sound subsystem. The allocated number is returned on succes. On failure a negative error code is returned.

# register\_sound\_mixer

## Name

`register_sound_mixer` — register a mixer device

## Synopsis

```
int register_sound_mixer (struct file_operations * fops, int  
dev);
```

## Arguments

*fops*

File operations for the driver

*dev*

Unit number to allocate

## Description

Allocate a mixer device. Unit is the number of the mixer requested. Pass -1 to request the next free mixer unit. On success the allocated number is returned, on failure a negative error code is returned.

# register\_sound\_midi

## Name

`register_sound_midi` — register a midi device

## Synopsis

```
int register_sound_midi (struct file_operations * fops, int  
dev);
```

## Arguments

*fops*

File operations for the driver

*dev*

Unit number to allocate

## Description

Allocate a midi device. Unit is the number of the midi device requested. Pass -1 to request the next free midi unit. On success the allocated number is returned, on failure a negative error code is returned.

# register\_sound\_dsp

## Name

`register_sound_dsp` — register a DSP device

## Synopsis

```
int register_sound_dsp (struct file_operations * fops, int dev);
```

## Arguments

*fops*

File operations for the driver

*dev*

Unit number to allocate

## Description

Allocate a DSP device. Unit is the number of the DSP requested. Pass -1 to request the next free DSP unit. On success the allocated number is returned, on failure a negative error code is returned.

This function allocates both the audio and dsp device entries together and will always allocate them as a matching pair - eg dsp3/audio3

# register\_sound\_synth

## Name

`register_sound_synth` — register a synth device

## Synopsis

```
int register_sound_synth (struct file_operations * fops, int  
dev);
```

## Arguments

*fops*

File operations for the driver

*dev*

Unit number to allocate

## Description

Allocate a synth device. Unit is the number of the synth device requested. Pass -1 to request the next free synth unit. On success the allocated number is returned, on failure a negative error code is returned.

# unregister\_sound\_special

## Name

`unregister_sound_special` — unregister a special sound device

## Synopsis

```
void unregister_sound_special (int unit);
```

## Arguments

*unit*

unit number to allocate

## Description

Release a sound device that was allocated with `register_sound_special`. The unit passed is the return value from the register function.

# unregister\_sound\_mixer

## Name

`unregister_sound_mixer` — unregister a mixer

## Synopsis

```
void unregister_sound_mixer (int unit);
```

## Arguments

*unit*

unit number to allocate

## Description

Release a sound device that was allocated with `register_sound_mixer`. The unit passed is the return value from the register function.

# unregister\_sound\_midi

## Name

`unregister_sound_midi` — unregister a midi device

## Synopsis

```
void unregister_sound_midi (int unit);
```



## Arguments

*unit*

unit number to allocate

## Description

Release a sound device that was allocated with `register_sound_midi`. The `unit` passed is the return value from the register function.

# unregister\_sound\_dsp

## Name

`unregister_sound_dsp` — unregister a DSP device

## Synopsis

```
void unregister_sound_dsp (int unit);
```

## Arguments

*unit*

unit number to allocate

## Description

Release a sound device that was allocated with `register_sound_dsp`. The `unit` passed is the return value from the register function.

Both of the allocated units are released together automatically.

# unregister\_sound\_synth

## Name

`unregister_sound_synth` — unregister a synth device

## Synopsis

```
void unregister_sound_synth (int unit);
```

## Arguments

*unit*

unit number to allocate

## Description

Release a sound device that was allocated with `register_sound_synth`. The unit passed is the return value from the register function.

# Chapter 17. USB Devices

## usb\_register

### Name

`usb_register` — register a USB driver

### Synopsis

```
int usb_register (struct usb_driver * new_driver);
```

### Arguments

*new\_driver*

USB operations for the driver

### Description

Registers a USB driver with the USB core. The list of unattached interfaces will be rescanned whenever a new driver is added, allowing the new driver to attach to any recognized devices. Returns a negative error code on failure and 0 on success.

## usb\_scan\_devices

### Name

`usb_scan_devices` — scans all unclaimed USB interfaces

### Synopsis

```
void usb_scan_devices ( void );
```

### Arguments

*void*

no arguments

### Description

Goes through all unclaimed USB interfaces, and offers them to all registered USB drivers through the 'probe' function. This will automatically be called after `usb_register` is called. It is called by some of the USB subsystems after one of their subdrivers are registered.

## usb\_deregister

### Name

`usb_deregister` — unregister a USB driver

### Synopsis

```
void usb_deregister (struct usb_driver * driver);
```

### Arguments

*driver*

USB operations of the driver to unregister

### Description

Unlinks the specified driver from the internal USB driver list.

## usb\_alloc\_bus

### Name

`usb_alloc_bus` — creates a new USB host controller structure

### Synopsis

```
struct usb_bus * usb_alloc_bus (struct usb_operations * op);
```

### Arguments

*op*

pointer to a struct `usb_operations` that this bus structure should use

### Description

Creates a USB host controller bus structure with the specified `usb_operations` and initializes all the necessary internal objects. (For use only by USB Host Controller Drivers.)

If no memory is available, `NULL` is returned.

The caller should call `usb_free_bus` when it is finished with the structure.

## usb\_free\_bus

### Name

`usb_free_bus` — frees the memory used by a bus structure

### Synopsis

```
void usb_free_bus (struct usb_bus * bus);
```

### Arguments

*bus*

pointer to the bus to free

### Description

(For use only by USB Host Controller Drivers.)



## usb\_register\_bus

### Name

`usb_register_bus` — registers the USB host controller with the usb core

### Synopsis

```
void usb_register_bus (struct usb_bus * bus);
```

### Arguments

*bus*

pointer to the bus to register

### Description

(For use only by USB Host Controller Drivers.)

## usb\_deregister\_bus

### Name

`usb_deregister_bus` — deregisters the USB host controller

### Synopsis

```
void usb_deregister_bus (struct usb_bus * bus);
```

### Arguments

*bus*

pointer to the bus to deregister

### Description

(For use only by USB Host Controller Drivers.)

## usb\_match\_id

### Name

`usb_match_id` — find first `usb_device_id` matching device or interface

### Synopsis

```
const struct usb_device_id * usb_match_id (struct usb_device *  
dev, struct usb_interface * interface, const struct  
usb_device_id * id);
```

### Arguments

*dev*

the device whose descriptors are considered when matching

*interface*

the interface of interest

*id*

array of `usb_device_id` structures, terminated by zero entry

## Description

`usb_match_id` searches an array of `usb_device_id`'s and returns the first one matching the device or interface, or null. This is used when binding (or rebinding) a driver to an interface. Most USB device drivers will use this indirectly, through the usb core, but some layered driver frameworks use it directly. These device tables are exported with `MODULE_DEVICE_TABLE`, through `modutils` and “`modules.usbmap`”, to support the driver loading functionality of USB hotplugging.

## What Matches

The “`match_flags`” element in a `usb_device_id` controls which members are used. If the corresponding bit is set, the value in the `device_id` must match its corresponding member in the device or interface descriptor, or else the `device_id` does not match.

“`driver_info`” is normally used only by device drivers, but you can create a wildcard “matches anything” `usb_device_id` as a driver's “`modules.usbmap`” entry if you provide an id with only a nonzero “`driver_info`” field. If you do this, the USB device driver's `probe` routine should use additional intelligence to decide whether to bind to the specified interface.

## What Makes Good `usb_device_id` Tables

The match algorithm is very simple, so that intelligence in driver selection must come from smart driver id records. Unless you have good reasons to use another selection policy, provide match elements only in related groups, and order match specifiers from specific to general. Use the macros provided for that purpose if you can.

The most specific match specifiers use device descriptor data. These are commonly used with product-specific matches; the `USB_DEVICE` macro lets you provide vendor and product IDs, and you can also match against ranges of product revisions. These are widely used for devices with application or vendor specific `bDeviceClass` values.

Matches based on device class/subclass/protocol specifications are slightly more general; use the `USB_DEVICE_INFO` macro, or its siblings. These are used with single-function devices where `bDeviceClass` doesn't specify that each interface has its own class.

Matches based on interface class/subclass/protocol are the most general; they let drivers bind to any interface on a multiple-function device. Use the `USB_INTERFACE_INFO` macro, or its siblings, to match class-per-interface style devices (as recorded in `bDeviceClass`).

Within those groups, remember that not all combinations are meaningful. For example, don't give a product version range without vendor and product IDs; or specify a protocol without its associated class and subclass.

## **usb\_alloc\_urb**

### **Name**

`usb_alloc_urb` — creates a new urb for a USB driver to use

### **Synopsis**

```
urb_t * usb_alloc_urb (int iso_packets);
```

## Arguments

*iso\_packets*

number of iso packets for this urb

## Description

Creates an urb for the USB driver to use and returns a pointer to it. If no memory is available, NULL is returned.

If the driver want to use this urb for interrupt, control, or bulk endpoints, pass '0' as the number of iso packets.

The driver should call `usb_free_urb` when it is finished with the urb.

## usb\_free\_urb

### Name

`usb_free_urb` — frees the memory used by a urb

### Synopsis

```
void usb_free_urb (urb_t* urb);
```

## Arguments

*urb*

pointer to the urb to free

## Description

If an urb is created with a call to `usb_create_urb` it should be cleaned up with a call to `usb_free_urb` when the driver is finished with it.

# usb\_control\_msg

## Name

`usb_control_msg` — Builds a control urb, sends it off and waits for completion

## Synopsis

```
int usb_control_msg (struct usb_device * dev, unsigned int pipe,
__u8 request, __u8 requesttype, __u16 value, __u16 index, void *
data, __u16 size, int timeout);
```

## Arguments

*dev*

pointer to the usb device to send the message to

*pipe*

endpoint “pipe” to send the message to

*request*

USB message request value

*requesttype*

USB message request type value

*value*

USB message value

*index*

USB message index value

*data*

pointer to the data to send

*size*

length in bytes of the data to send

*timeout*

time to wait for the message to complete before timing out (if 0 the wait is forever)



## Description

This function sends a simple control message to a specified endpoint and waits for the message to complete, or timeout.

If successful, it returns 0, otherwise a negative error number.

Don't use this function from within an interrupt context, like a bottom half handler. If you need an asynchronous message, or need to send a message from within interrupt context, use `usb_submit_urb`.

## usb\_bulk\_msg

### Name

`usb_bulk_msg` — Builds a bulk urb, sends it off and waits for completion

### Synopsis

```
int usb_bulk_msg (struct usb_device * usb_dev, unsigned int
pipe, void * data, int len, int * actual_length, int timeout);
```

## Arguments

*usb\_dev*

pointer to the usb device to send the message to

*pipe*

endpoint “pipe” to send the message to

*data*

pointer to the data to send

*len*

length in bytes of the data to send

*actual\_length*

pointer to a location to put the actual length transferred in bytes

*timeout*

time to wait for the message to complete before timing out (if 0 the wait is forever)

## Description

This function sends a simple bulk message to a specified endpoint and waits for the message to complete, or timeout.

If successful, it returns 0, otherwise a negative error number. The number of actual bytes transferred will be placed in the *actual\_length* parameter.

Don't use this function from within an interrupt context, like a bottom half handler. If you need an asynchronous message, or need to send a message from within interrupt context, use `usb_submit_urb`



# Chapter 18. 16x50 UART Driver

## register\_serial

### Name

`register_serial` — configure a 16x50 serial port at runtime

### Synopsis

```
int register_serial (struct serial_struct * req);
```

### Arguments

*req*

request structure

### Description

Configure the serial port specified by the request. If the port exists and is in use an error is returned. If the port is not currently in the table it is added.

The port is then probed and if neccessary the IRQ is autodetected If this fails an error is returned.

On success the port is ready to use and the line number is returned.

## unregister\_serial

### Name

`unregister_serial` — deconfigure a 16x50 serial port

### Synopsis

```
void unregister_serial (int line);
```

### Arguments

*line*

line to deconfigure

### Description

The port specified is deconfigured and its resources are freed. Any user of the port is disconnected as if carrier was dropped. Line is the port number returned by `register_serial`.



# Chapter 19. Z85230 Support Library

## z8530\_interrupt

### Name

`z8530_interrupt` — Handle an interrupt from a Z8530

### Synopsis

```
void z8530_interrupt (int irq, void * dev_id, struct pt_regs *  
regs);
```

### Arguments

*irq*

Interrupt number

*dev\_id*

The Z8530 device that is interrupting.

*regs*

unused

## Description

A Z85[2]30 device has stuck its hand in the air for attention. We scan both the channels on the chip for events and then call the channel specific call backs for each channel that has events. We have to use callback functions because the two channels can be in different modes.

Locking is done for the handlers. Note that locking is done at the chip level (the 5uS delay issue is per chip not per channel). `c->lock` for both channels points to `dev->lock`

## z8530\_sync\_open

### Name

`z8530_sync_open` — Open a Z8530 channel for PIO

### Synopsis

```
int z8530_sync_open (struct net_device * dev, struct  
z8530_channel * c);
```



## Arguments

*dev*

The network interface we are using

*c*

The Z8530 channel to open in synchronous PIO mode

## Description

Switch a Z8530 into synchronous mode without DMA assist. We raise the RTS/DTR and commence network operation.

# z8530\_sync\_close

## Name

z8530\_sync\_close — Close a PIO Z8530 channel

## Synopsis

```
int z8530_sync_close (struct net_device * dev, struct  
z8530_channel * c);
```

## Arguments

*dev*

Network device to close

*c*

Z8530 channel to disassociate and move to idle

## Description

Close down a Z8530 interface and switch its interrupt handlers to discard future events.

# z8530\_sync\_dma\_open

## Name

z8530\_sync\_dma\_open — Open a Z8530 for DMA I/O

## Synopsis

```
int z8530_sync_dma_open (struct net_device * dev, struct  
z8530_channel * c);
```

## Arguments

*dev*

The network device to attach

*c*

The Z8530 channel to configure in sync DMA mode.

## Description

Set up a Z85x30 device for synchronous DMA in both directions. Two ISA DMA channels must be available for this to work. We assume ISA DMA driven I/O and PC limits on access.

# z8530\_sync\_dma\_close

## Name

`z8530_sync_dma_close` — Close down DMA I/O

## Synopsis

```
int z8530_sync_dma_close (struct net_device * dev, struct  
z8530_channel * c);
```

## Arguments

*dev*

Network device to detach

*c*

Z8530 channel to move into discard mode

## Description

Shut down a DMA mode synchronous interface. Halt the DMA, and free the buffers.

# z8530\_sync\_txdma\_open

## Name

z8530\_sync\_txdma\_open — Open a Z8530 for TX driven DMA

## Synopsis

```
int z8530_sync_txdma_open (struct net_device * dev, struct  
z8530_channel * c);
```

## Arguments

*dev*

The network device to attach

*c*

The Z8530 channel to configure in sync DMA mode.

## Description

Set up a Z85x30 device for synchronous DMA transmission. One ISA DMA channel must be available for this to work. The receive side is run in PIO mode, but then it has the bigger FIFO.

## **z8530\_sync\_txdma\_close**

### **Name**

`z8530_sync_txdma_close` — Close down a TX driven DMA channel

## Synopsis

```
int z8530_sync_txdma_close (struct net_device * dev, struct  
z8530_channel * c);
```

## Arguments

*dev*

Network device to detach

*c*

Z8530 channel to move into discard mode

## Description

Shut down a DMA/PIO split mode synchronous interface. Halt the DMA, and free the buffers.

## **z8530\_describe**

### Name

`z8530_describe` — Uniformly describe a Z8530 port

## Synopsis

```
void z8530_describe (struct z8530_dev * dev, char * mapping,  
unsigned long io);
```

## Arguments

*dev*

Z8530 device to describe

*mapping*

string holding mapping type (eg “I/O” or “Mem”)

*io*

the port value in question

## Description

Describe a Z8530 in a standard format. We must pass the I/O as the port offset isn't predictable. The main reason for this function is to try and get a common format of report.

## **z8530\_init**

### **Name**

`z8530_init` — Initialise a Z8530 device

### **Synopsis**

```
int z8530_init (struct z8530_dev * dev);
```

### **Arguments**

*dev*

Z8530 device to initialise.

### **Description**

Configure up a Z8530/Z85C30 or Z85230 chip. We check the device is present, identify the type and then program it to hopefully keep quite and behave. This matters a lot, a Z8530 in the wrong state will sometimes get into stupid modes generating 10Khz interrupt streams and the like.

We set the interrupt handler up to discard any events, in case we get them during reset or setp.

Return 0 for success, or a negative value indicating the problem in errno form.



## **z8530\_shutdown**

### **Name**

`z8530_shutdown` — Shutdown a Z8530 device

### **Synopsis**

```
int z8530_shutdown (struct z8530_dev * dev);
```

### **Arguments**

*dev*

The Z8530 chip to shutdown

### **Description**

We set the interrupt handlers to silence any interrupts. We then reset the chip and wait 100uS to be sure the reset completed. Just in case the caller then tries to do stuff.

This is called without the lock held

## z8530\_channel\_load

### Name

z8530\_channel\_load — Load channel data

### Synopsis

```
int z8530_channel_load (struct z8530_channel * c, u8 * rtable);
```

### Arguments

*c*

Z8530 channel to configure

*rtable*

table of register, value pairs

### FIXME

ioctl to allow user uploaded tables

Load a Z8530 channel up from the system data. We use +16 to indicate the “prime” registers. The value 255 terminates the table.

## **z8530\_null\_rx**

### **Name**

`z8530_null_rx` — Discard a packet

### **Synopsis**

```
void z8530_null_rx (struct z8530_channel * c, struct sk_buff *  
skb);
```

### **Arguments**

*c*

The channel the packet arrived on

*skb*

The buffer

### **Description**

We point the receive handler at this function when idle. Instead of syncppp processing the frames we get to throw them away.

## z8530\_queue\_xmit

### Name

z8530\_queue\_xmit — Queue a packet

### Synopsis

```
int z8530_queue_xmit (struct z8530_channel * c, struct sk_buff *  
skb);
```

### Arguments

*c*

The channel to use

*skb*

The packet to kick down the channel

### Description

Queue a packet for transmission. Because we have rather hard to hit interrupt latencies for the Z85230 per packet even in DMA mode we do the flip to DMA buffer if needed here not in the IRQ.

Called from the network code. The lock is not held at this point.

## z8530\_get\_stats

### Name

z8530\_get\_stats — Get network statistics

### Synopsis

```
struct net_device_stats * z8530_get_stats (struct z8530_channel  
* c);
```

### Arguments

*c*

The channel to use

### Description

Get the statistics block. We keep the statistics in software as the chip doesn't do it for us.

Locking is ignored here - we could lock for a copy but its not likely to be that big an issue



# Chapter 20. Frame Buffer Library

The frame buffer drivers depend heavily on four data structures. These structures are declared in `include/linux/fb.h`. They are `fb_info`, `fb_var_screeninfo`, `fb_fix_screeninfo` and `fb_monospecs`. The last three can be made available to and from userland.

`fb_info` defines the current state of a particular video card. Inside `fb_info`, there exists a `fb_ops` structure which is a collection of needed functions to make `fbdev` and `fbcon` work. `fb_info` is only visible to the kernel.

`fb_var_screeninfo` is used to describe the features of a video card that are user defined. With `fb_var_screeninfo`, things such as depth and the resolution may be defined.

The next structure is `fb_fix_screeninfo`. This defines the properties of a card that are created when a mode is set and can't be changed otherwise. A good example of this is the start of the frame buffer memory. This "locks" the address of the frame buffer memory, so that it cannot be changed or moved.

The last structure is `fb_monospecs`. In the old API, there was little importance for `fb_monospecs`. This allowed for forbidden things such as setting a mode of 800x600 on a fix frequency monitor. With the new API, `fb_monospecs` prevents such things, and if used correctly, can prevent a monitor from being cooked. `fb_monospecs` will not be useful until kernels 2.5.x.

## 20.1. Frame Buffer Memory

### `register_framebuffer`

#### **Name**

`register_framebuffer` — registers a frame buffer device

## Synopsis

```
int register_framebuffer (struct fb_info * fb_info);
```

## Arguments

*fb\_info*

frame buffer info structure

## Description

Registers a frame buffer device *fb\_info*.

Returns negative errno on error, or zero for success.

# unregister\_framebuffer

## Name

`unregister_framebuffer` — releases a frame buffer device



## Synopsis

```
int unregister_framebuffer (struct fb_info * fb_info);
```

## Arguments

*fb\_info*

frame buffer info structure

## Description

Unregisters a frame buffer device *fb\_info*.

Returns negative errno on error, or zero for success.

## 20.2. Frame Buffer Console

### fbcon\_redraw\_clear

#### Name

fbcon\_redraw\_clear — clear area of the screen

## Synopsis

```
void fbcon_redraw_clear (struct vc_data * conp, struct display *  
p, int sy, int sx, int height, int width);
```

## Arguments

*conp*

structure pointing to current active virtual console

*p*

display structure

*sy*

starting Y coordinate

*sx*

starting X coordinate

*height*

height of area to clear

*width*

width of area to clear

## Description

Clears a specified area of the screen. All dimensions are in pixels.

## fbcon\_redraw\_bmove

### Name

`fbcon_redraw_bmove` — copy area of screen to another area

### Synopsis

```
void fbcon_redraw_bmove (struct display * p, int sy, int sx, int  
dy, int dx, int h, int w);
```

### Arguments

*p*

display structure

*sy*

origin Y coordinate

*sx*

origin X coordinate

*dy*

destination Y coordinate

*dx*

destination X coordinate

*h*

height of area to copy

*w*

width of area to copy

## **Description**

Copies an area of the screen to another area of the same screen. All dimensions are in pixels.

Note that this function cannot be used together with `ypan` or `ywrap`.

## 20.3. Frame Buffer Colormap

### **fb\_alloc\_cmap**

#### **Name**

`fb_alloc_cmap` — allocate a colormap

#### **Synopsis**

```
int fb_alloc_cmap (struct fb_cmap * cmap, int len, int transp);
```

#### **Arguments**

*cmap*

frame buffer colormap structure

*len*

length of *cmap*

*transp*

boolean, 1 if there is transparency, 0 otherwise

## Description

Allocates memory for a colormap *cmap*. *len* is the number of entries in the palette.

Returns -1 *errno* on error, or zero on success.

## fb\_copy\_cmap

### Name

`fb_copy_cmap` — copy a colormap

### Synopsis

```
void fb_copy_cmap (struct fb_cmap * from, struct fb_cmap * to,  
int fsfromto);
```

### Arguments

*from*

frame buffer colormap structure

*to*

frame buffer colormap structure

*fsfromto*

determine copy method

## Description

Copy contents of colormap from *from* to *to*.

**0**

memcpy function

**1**

`copy_from_user` function to copy from userspace

**2**

`copy_to_user` function to copy to userspace

## fb\_get\_cmap

### Name

`fb_get_cmap` — get a colormap

## Synopsis

```
int fb_get_cmap (struct fb_cmap * cmap, int kspc, int  
(*getcolreg) (u_int, u_int *, u_int *, u_int *, u_int *, struct  
fb_info *), struct fb_info * info);
```

## Arguments

*cmap*

frame buffer colormap

*kspc*

boolean, 0 copy local, 1 put\_user function

*getcolreg*

pointer to a function to get a color register

*info*

frame buffer info structure

## Description

Get a colormap *cmap* for a screen of device *info*.

Returns negative errno on error, or zero on success.



## fb\_set\_cmap

### Name

`fb_set_cmap` — set the colormap

### Synopsis

```
int fb_set_cmap (struct fb_cmap * cmap, int kspc, int  
(*setcolreg) (u_int, u_int, u_int, u_int, u_int, struct fb_info  
*), struct fb_info * info);
```

### Arguments

*cmap*

frame buffer colormap structure

*kspc*

boolean, 0 copy local, 1 `get_user` function

*setcolreg*

– undescribed –

*info*

frame buffer info structure

## Description

Sets the colormap *cmap* for a screen of device *info*.

Returns negative *errno* on error, or zero on success.

## fb\_default\_cmap

### Name

`fb_default_cmap` — get default colormap

### Synopsis

```
struct fb_cmap * fb_default_cmap (int len);
```

### Arguments

*len*

size of palette for a depth

## Description

Gets the default colormap for a specific screen depth. *len* is the size of the palette for a particular screen depth.

Returns pointer to a frame buffer colormap structure.

## fb\_invert\_cmaps

### Name

`fb_invert_cmaps` — invert all defaults colormaps

### Synopsis

```
void fb_invert_cmaps ( void );
```

### Arguments

*void*

no arguments

## Description

Invert all default colormaps.

## 20.4. Frame Buffer Generic Functions

### fbgen\_get\_fix

#### Name

`fbgen_get_fix` — get fixed part of display

#### Synopsis

```
int fbgen_get_fix (struct fb_fix_screeninfo * fix, int con,  
struct fb_info * info);
```

#### Arguments

*fix*

fb\_fix\_screeninfo structure

*con*

virtual console number

*info*

frame buffer info structure

## Description

Get the fixed information part of the display and place it into *fix* for virtual console *con* on device *info*.

Returns negative *errno* on error, or zero on success.

## fbgen\_get\_var

### Name

`fbgen_get_var` — get user defined part of display

### Synopsis

```
int fbgen_get_var (struct fb_var_screeninfo * var, int con,  
struct fb_info * info);
```

## Arguments

*var*

fb\_var\_screeninfo structure

*con*

virtual console number

*info*

frame buffer info structure

## Description

Get the user defined part of the display and place it into *var* for virtual console *con* on device *info*.

Returns negative errno on error, or zero for success.

## fbgen\_set\_var

### Name

fbgen\_set\_var — set the user defined part of display

## Synopsis

```
int fbgen_set_var (struct fb_var_screeninfo * var, int con,  
struct fb_info * info);
```

## Arguments

*var*

fb\_var\_screeninfo user defined part of the display

*con*

virtual console number

*info*

frame buffer info structure

## Description

Set the user defined part of the display as dictated by *var* for virtual console *con* on device *info*.

Returns negative errno on error, or zero for success.

# fbgen\_get\_cmap

## Name

`fbgen_get_cmap` — get the colormap

## Synopsis

```
int fbgen_get_cmap (struct fb_cmap * cmap, int kspc, int con,  
struct fb_info * info);
```

## Arguments

*cmap*

frame buffer colormap structure

*kspc*

boolean, 0 copy local, 1 put\_user function

*con*

virtual console number

*info*

frame buffer info structure



## Description

Gets the colormap for virtual console *con* and places it into *cmap* for device *info*.

Returns negative *errno* on error, or zero for success.

# fbgen\_set\_cmap

## Name

`fbgen_set_cmap` — set the colormap

## Synopsis

```
int fbgen_set_cmap (struct fb_cmap * cmap, int kspc, int con,  
struct fb_info * info);
```

## Arguments

*cmap*

frame buffer colormap structure

*kspc*

boolean, 0 copy local, 1 `get_user` function

*con*

virtual console number

*info*

frame buffer info structure

## Description

Sets the colormap *cmap* for virtual console *con* on device *info*.

Returns negative *errno* on error, or zero for success.

# fbgen\_pan\_display

## Name

`fbgen_pan_display` — pan or wrap the display

## Synopsis

```
int fbgen_pan_display (struct fb_var_screeninfo * var, int con,  
struct fb_info * info);
```

## Arguments

*var*

frame buffer user defined part of display

*con*

virtual console number

*info*

frame buffer info structure

## Description

Pan or wrap virtual console *con* for device *info*.

This call looks only at xoffset, yoffset and the FB\_VMODE\_YWRAP flag in *var*.

Returns negative errno on error, or zero for success.

## fbgen\_do\_set\_var

### Name

fbgen\_do\_set\_var — change the video mode

## Synopsis

```
int fbgen_do_set_var (struct fb_var_screeninfo * var, int
isactive, struct fb_info_gen * info);
```

## Arguments

*var*

frame buffer user defined part of display

*isactive*

boolean, 0 inactive, 1 active

*info*

generic frame buffer info structure

## Description

Change the video mode settings for device *info*. If *isactive* is non-zero, the changes will be activated immediately.

Return negative errno on error, or zero for success.

# fbgen\_set\_disp

## Name

`fbgen_set_disp` — set generic display

## Synopsis

```
void fbgen_set_disp (int con, struct fb_info_gen * info);
```

## Arguments

*con*

virtual console number

*info*

generic frame buffer info structure

## Description

Sets a display on virtual console *con* for device *info*.

# fbgen\_install\_cmap

## Name

`fbgen_install_cmap` — install the current colormap

## Synopsis

```
void fbgen_install_cmap (int con, struct fb_info_gen * info);
```

## Arguments

*con*

virtual console number

*info*

generic frame buffer info structure

## Description

Installs the current colormap for virtual console *con* on device *info*.

# fbgen\_update\_var

## Name

`fbgen_update_var` — update user defined part of display

## Synopsis

```
int fbgen_update_var (int con, struct fb_info * info);
```

## Arguments

*con*

virtual console number

*info*

frame buffer info structure

## Description

Updates the user defined part of the display ('var' structure) on virtual console *con* for device *info*. This function is called by `fbcon.c`.

Returns negative `errno` on error, or zero for success.

# fbgen\_switch

## Name

`fbgen_switch` — switch to a different virtual console.

## Synopsis

```
int fbgen_switch (int con, struct fb_info * info);
```

## Arguments

*con*

virtual console number

*info*

frame buffer info structure

## Description

Switch to virtual console *con* on device *info*.

Returns zero.



# fbgen\_blank

## Name

`fbgen_blank` — blank the screen

## Synopsis

```
void fbgen_blank (int blank, struct fb_info * info);
```

## Arguments

*blank*

boolean, 0 unblank, 1 blank

*info*

frame buffer info structure

## Description

Blank the screen on device *info*.

## 20.5. Frame Buffer Video Mode Database

### fb\_find\_mode

#### Name

`fb_find_mode` — finds a valid video mode

#### Synopsis

```
int __init fb_find_mode (struct fb_var_screeninfo * var, struct
fb_info * info, const char * mode_option, const struct
fb_videomode * db, unsigned int dbsize, const struct
fb_videomode * default_mode, unsigned int default_bpp);
```

#### Arguments

*var*

frame buffer user defined part of display

*info*

frame buffer info structure

*mode\_option*

string video mode to find

*db*

video mode database

*dbsize*

size of *db*

*default\_mode*

default video mode to fall back to

*default\_bpp*

default color depth in bits per pixel

## Description

Finds a suitable video mode, starting with the specified mode in *mode\_option* with fallback to *default\_mode*. If *default\_mode* fails, all modes in the video mode database will be tried.

Valid mode specifiers for *mode\_option*:

<xres>x<yres>[-<bpp>][@<refresh>] or <name>[-<bpp>][@<refresh>]

with <xres>, <yres>, <bpp> and <refresh> decimal numbers and <name> a string.

## NOTE

The passed struct *var* is *\_not\_* cleared! This allows you to supply values for e.g. the *grayscale* and *accel\_flags* fields.

Returns zero for failure, 1 if using specified *mode\_option*, 2 if using specified *mode\_option* with an ignored refresh rate, 3 if default mode is used, 4 if fall back to any valid mode.

## **\_\_fb\_try\_mode**

### **Name**

`__fb_try_mode` — test a video mode

### **Synopsis**

```
int __fb_try_mode (struct fb_var_screeninfo * var, struct  
fb_info * info, const struct fb_videomode * mode, unsigned int  
bpp);
```

### **Arguments**

*var*

frame buffer user defined part of display

*info*

frame buffer info structure

*mode*

frame buffer video mode structure

*bpp*

color depth in bits per pixel

## Description

Tries a video mode to test it's validity for device *info*.

Returns 1 on success.

## 20.6. Frame Buffer Macintosh Video Mode Database

### console\_getmode

#### Name

`console_getmode` — get current mode

#### Synopsis

```
int console_getmode (struct vc_mode * mode);
```

## Arguments

*mode*

virtual console mode structure

## Description

Populates *mode* with the current mode held in the global `display_info` structure.

Note, this function is only for XPMAC compatibility.

Returns zero.

# console\_setmode

## Name

`console_setmode` — sets current console mode

## Synopsis

```
int console_setmode (struct vc_mode * mode, int doit);
```

## Arguments

*mode*

virtual console mode structure

*doit*

boolean, 0 test mode, 1 test and activate mode

## Description

Sets *mode* for all virtual consoles if *doit* is non-zero, otherwise, test a mode for validity.

Note, this function is only for XPMAC compatibility.

Returns negative `errno` on error, or zero for success.

# console\_setcmap

## Name

`console_setcmap` — sets palette color map for console

## Synopsis

```
int console_setcmap (int n_entries, unsigned char * red,
```

```
unsigned char * green, unsigned char * blue);
```

## Arguments

*n\_entries*

number of entries in the palette (max 16)

*red*

value for red component of palette

*green*

value for green component of palette

*blue*

value for blue component of palette

## Description

Sets global `palette_cmap` structure and activates the palette on the current console.

Note, this function is only for XPMAC compatibility.

Returns negative `errno` on error, or zero for success.



# console\_powermode

## Name

`console_powermode` — sets monitor power mode

## Synopsis

```
int console_powermode (int mode);
```

## Arguments

*mode*

power state to set

## Description

Sets power state as dictated by *mode*.

Note that this function is only for XPMAC compatibility and doesn't do much.

Returns 0 for `VC_POWERMODE_INQUIRY`, `-EINVAL` for VESA power settings, or `-ENXIO` on failure.

## mac\_vmode\_to\_var

### Name

`mac_vmode_to_var` — converts vmode/cmode pair to var structure

### Synopsis

```
int mac_vmode_to_var (int vmode, int cmode, struct  
fb_var_screeninfo * var);
```

### Arguments

*vmode*

MacOS video mode

*cmode*

MacOS color mode

*var*

frame buffer video mode structure

### Description

Converts a MacOS vmode/cmode pair to a frame buffer video mode structure.

Returns negative `errno` on error, or zero for success.

## **mac\_var\_to\_vmode**

### **Name**

`mac_var_to_vmode` — convert `var` structure to MacOS `vmode/cmode` pair

### **Synopsis**

```
int mac_var_to_vmode (const struct fb_var_screeninfo * var, int
* vmode, int * cmode);
```

### **Arguments**

*var*

frame buffer video mode structure

*vmode*

MacOS video mode

*cmode*

MacOS color mode

## Description

Converts a frame buffer video mode structure to a MacOS vmode/cmode pair.

Returns negative errno on error, or zero for success.

# mac\_map\_monitor\_sense

## Name

`mac_map_monitor_sense` — Convert monitor sense to vmode

## Synopsis

```
int mac_map_monitor_sense (int sense);
```

## Arguments

*sense*

Macintosh monitor sense number

## Description

Converts a Macintosh monitor sense number to a MacOS vmode number.

Returns MacOS vmode video mode number.

# mac\_find\_mode

## Name

`mac_find_mode` — find a video mode

## Synopsis

```
int __init mac_find_mode (struct fb_var_screeninfo * var, struct
fb_info * info, const char * mode_option, unsigned int
default_bpp);
```

## Arguments

*var*

frame buffer user defined part of display

*info*

frame buffer info structure

*mode\_option*

video mode name (see `mac_modedb[]`)

*default\_bpp*

default color depth in bits per pixel

## Description

Finds a suitable video mode. Tries to set mode specified by *mode\_option*. If the name of the wanted mode begins with 'mac', the Mac video mode database will be used, otherwise it will fall back to the standard video mode database.

## Note

Function marked as `__init` and can only be used during system boot.

Returns error code from `fb_find_mode` (see `fb_find_mode` function).

## 20.7. Frame Buffer Fonts

### **fbcon\_find\_font**

#### **Name**

`fbcon_find_font` — find a font

#### **Synopsis**

```
struct fbcon_font_desc * fbcon_find_font (char * name);
```

#### **Arguments**

*name*

string name of a font

#### **Description**

Find a specified font with string name *name*.

Returns `NULL` if no font found, or a pointer to the specified font.

## fbcon\_get\_default\_font

### Name

`fbcon_get_default_font` — get default font

### Synopsis

```
struct fbcon_font_desc * fbcon_get_default_font (int xres, int
yres);
```

### Arguments

*xres*

screen size of X

*yres*

screen size of Y

### Description

Get the default font for a specified screen size. Dimensions are in pixels.

Returns `NULL` if no font is found, or a pointer to the chosen font.