

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

1. The Linux VFS.....	10
The Directory Cache	10
d_invalidate.....	10
d_find_alias.....	10
prune_dcache	11
shrink_dcache_sb	12
have_submounts	12
shrink_dcache_parent	13
d_alloc	13
d_instantiate.....	14
d_alloc_root.....	15
d_lookup	16
d_validate.....	16
d_delete.....	17
d_rehash.....	18
d_move	18
__d_path	19
is_subdir	20
find_inode_number	21
d_drop.....	22
d_add	22
dget	23
d_unhashed.....	24
Inode Handling.....	24
__mark_inode_dirty	25
write_inode_now.....	25
clear_inode	26
invalidate_inodes	26
get_empty_inode	27
iunique	28
insert_inode_hash	29
remove_inode_hash	29
iput.....	30
bmap	30
update_atime.....	31
make_bad_inode.....	32
is_bad_inode.....	32
Registration and Superblocks.....	33

register_filesystem	33
unregister_filesystem	34
__wait_on_super	34
get_super	35
get_empty_super	36
2. Linux Networking	37
Socket Buffer Functions	37
skb_queue_empty	37
skb_get	37
kfree_skb	38
skb_cloned	38
skb_shared	39
skb_unshare	40
skb_peek	40
skb_peek_tail	41
skb_queue_len	42
__skb_queue_head	42
skb_queue_head	43
__skb_queue_tail	44
skb_queue_tail	45
__skb_dequeue	45
skb_dequeue	46
skb_insert	47
skb_append	47
skb_unlink	48
__skb_dequeue_tail	49
skb_dequeue_tail	49
skb_put	50
skb_push	51
skb_pull	51
skb_headroom	52
skb_tailroom	53
skb_reserve	53
skb_trim	54
skb_orphan	55
skb_queue_purge	55
__skb_queue_purge	56
dev_alloc_skb	57
skb_cow	57
skb_over_panic	58

skb_under_panic	59
alloc_skb	60
__kfree_skb	61
skb_clone	61
skb_copy	62
skb_copy_expand	63
Socket Filter	64
sk_run_filter	64
3. Network device support.....	66
Driver Support	66
init_etherdev	66
dev_add_pack	66
dev_remove_pack	67
__dev_get_by_name	68
dev_get_by_name	68
dev_get	69
__dev_get_by_index	70
dev_get_by_index	70
dev_alloc_name	71
dev_alloc	72
netdev_state_change	73
dev_load	73
dev_open	74
dev_close	75
register_netdevice_notifier	75
unregister_netdevice_notifier	76
dev_queue_xmit	77
netif_rx	77
net_call_rx_atomic	78
register_gifconf	79
netdev_set_master	79
dev_set_promiscuity	80
dev_set_allmulti	81
dev_ioctl	81
dev_new_index	82
register_netdevice	83
netdev_finish_unregister	84
unregister_netdevice	84
8390 Based Network Cards	85
ei_open	85

ei_close	85
ei_interrupt	86
ethdev_init	87
NS8390_init.....	87
Synchronous PPP	88
sppp_input	88
sppp_close	89
sppp_open	90
sppp_reopen.....	90
sppp_change_mtu	91
sppp_do_ioctl	92
sppp_attach.....	92
sppp_detach	93
4. Module Loading	95
request_module	95
5. Hardware Interfaces	96
Interrupt Handling	96
disable_irq_nosync	96
disable_irq	96
enable_irq	97
probe_irq_mask	98
MTRR Handling	98
mtrr_add	98
mtrr_del	100
PCI Support Library.....	100
pci_find_slot	101
pci_find_device.....	101
pci_find_class	102
pci_find_parent_resource	103
pci_set_power_state.....	104
pci_enable_device	104
MCA Architecture.....	105
MCA Device Functions	105
mca_find_adapter	105
mca_find_unused_adapter.....	106
mca_read_stored_pos.....	107
mca_read_pos	107
mca_write_pos	108
mca_set_adapter_name	109
mca_set_adapter_procfn	110

mca_is_adapter_used	111
mca_mark_as_used	111
mca_mark_as_unused	112
mca_get_adapter_name.....	113
mca_isadapter	113
mca_isenabled.....	114
MCA Bus DMA	114
mca_enable_dma.....	114
mca_disable_dma.....	115
mca_set_dma_addr	116
mca_get_dma_addr	116
mca_set_dma_count.....	117
mca_get_dma_residue.....	118
mca_set_dma_io	118
mca_set_dma_mode.....	119
6. The Device File System.....	121
devfs_register	121
devfs_unregister	122
devfs_mk_symlink	123
devfs_mk_dir.....	124
devfs_find_handle	125
devfs_get_flags.....	126
devfs_get_maj_min	127
devfs_get_handle_from_inode	128
devfs_generate_path.....	128
devfs_get_ops.....	129
devfs_set_file_size	130
devfs_get_info.....	131
devfs_set_info	131
devfs_get_parent	132
devfs_get_first_child	133
devfs_get_next_sibling.....	133
devfs_auto_unregister	134
devfs_get_unregister_slave	134
devfs_register_chrdev	135
devfs_register_blkdev	136
devfs_unregister_chrdev	137
devfs_unregister_blkdev	138
7. Power Management	139
pm_register.....	139

pm_unregister.....	139
pm_unregister_all.....	140
pm_send	141
pm_send_all	142
pm_find	142
8. Miscellaneous Devices	144
misc_register	144
misc_deregister	144
9. Video4Linux	146
video_register_device.....	146
video_unregister_device.....	146
10. Sound Devices.....	148
register_sound_special	148
register_sound_mixer.....	148
register_sound_midi	149
register_sound_dsp.....	150
register_sound_synth.....	150
unregister_sound_special	151
unregister_sound_mixer	152
unregister_sound_midi	152
unregister_sound_dsp.....	153
unregister_sound_synth.....	154
11. 16x50 UART Driver	155
register_serial	155
unregister_serial	155
12. Z85230 Support Library	157
z8530_interrupt	157
z8530_sync_open	157
z8530_sync_close	158
z8530_sync_dma_open.....	159
z8530_sync_dma_close	160
z8530_sync_txdma_open.....	160
z8530_sync_txdma_close	161
z8530_describe	162
z8530_init.....	162
z8530_shutdown	163
z8530_channel_load.....	164
z8530_null_rx	164
z8530_queue_xmit	165

z8530_get_stats 166

Chapter 1. The Linux VFS

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.

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

dentree to complete

inode

inode to attach to this dentree

Description

Fill in inode information in the entry.

This turns negative dentrees 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 dentree

Synopsis

```
struct dentree * 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, unsigned int
hash, unsigned int len);
```

Arguments

dentry

The dentry alleged to be valid

dparent

The parent dentry

hash

Hash of the dentry

len

Length of the name

Description

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

NOTE

This function does `_not_` dereference the pointers before we have validated them. We can test the pointer values, but we must not actually use them until we have found a valid copy of the pointer in kernel space..

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 * vsmnt, struct
dentry * root, struct vfsmount * rootmnt, char * buffer, int buflen);
```

Arguments

dentry

dentry to report

vfsmnt

– undescribed –

root

– undescribed –

rootmnt

– undescribed –

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.

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.

`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.

Inode Handling

`__mark_inode_dirty`

Name `__mark_inode_dirty` — internal function

Synopsis

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

Arguments

inode

inode to mark

Description

Mark an inode as dirty. Callers should use `mark_inode_dirty`.

`write_inode_now`

Name `write_inode_now` — write an inode to disk

Synopsis

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

Arguments

inode

inode to write to disk

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.

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.

`__wait_on_super`

Name `__wait_on_super` — wait on a superblock

Synopsis

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

Arguments

sb

superblock to wait on

Description

Waits for a superblock to become unlocked and then returns. It does not take the lock. This is an internal function. See `wait_on_super`.

`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.

get_empty_super

Name `get_empty_super` — find empty superblocks

Synopsis

```
struct super_block * get_empty_super ( void );
```

Arguments

void

no arguments

Description

Find a superblock with no device assigned. A free superblock is found and returned. If necessary new superblocks are allocated. `NULL` is returned if there are insufficient resources to complete the request.

Chapter 2. Linux Networking

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_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);
```

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 in there is no free memory. Although this function allocates memory it can be called from an interrupt.

skb_cow

Name `skb_cow` — copy a buffer if need be

Synopsis

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

Arguments

skb

buffer to copy

headroom

needed headroom

Description

If the buffer passed lacks sufficient headroom or is a clone then it is copied and the additional headroom made available. If there is no free memory `NULL` is returned. The new buffer is returned if a copy was made (and the old one dropped a reference). The existing buffer is returned otherwise.

This function primarily exists to avoid making two copies when making a writable copy of a buffer and then growing the headroom.

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` — copy 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.

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

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.

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.

Chapter 3. Network device support

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.

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 “%td” 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. “%ld”, 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

```
void 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.

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 to be sure it remains unique.

register_netdevice

Name `register_netdevice` — register a network device

Synopsis

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

Arguments

dev

device to register

Description

Take a completed network device structure and add it to the kernel interfaces. A `NETDEV_REGISTER` message is sent to the `netdev` notifier chain. 0 is returned on success. A negative `errno` code is returned on a failure to set up the device, or if the name is a duplicate.

BUGS

The locking appears insufficient to guarantee two parallel registers will not get the same name.

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.

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_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 necessary. 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.

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 requeues 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 endonly.

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 4. 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.

Chapter 5. Hardware Interfaces

Interrupt Handling

disable_irq_nosync

Name `disable_irq_nosync` — disable an irq without waiting

Synopsis

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

Arguments

irq

Interrupt to disable

Description

Disable the selected interrupt line. Disables of an interrupt stack. 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. Disables of an interrupt stack. That is for two disables you need two enables. 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 interrupt handling on an `irq`

Synopsis

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

Arguments

irq

Interrupt to enable

Description

Re-enables the processing of interrupts on this IRQ line providing no `disable_irq` calls are now in effect.

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.

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_UNCACHEABLE - No caching

MTRR_TYPE_WRITEBACK - Write data back in bursts whenever

MTRR_TYPE_WRCOMB - Write data back soon but allow bursts

MTRR_TYPE_WRTROUGH - 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.

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

number of PCI slot in which desired PCI device resides

Description

Given a PCI bus and slot 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_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 vendor 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 that point.

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 that point.

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 power management state of a device.

Synopsis

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

Arguments

dev

PCI device for which PM is set

new_state

new power management statement (0 == D0, 3 == D3, etc.)

Description

Set power management state of a device. For transitions from state D3 it isn't as straightforward as one could assume since many devices forget their configuration space during wakeup. Returns old power state.

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.

MCA Architecture

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.

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 6. The Device File System

devfs_register

Name `devfs_register` — Register a device entry.

Synopsis

```
devfs_handle_t devfs_register (devfs_handle_t dir, const char * name,  
unsigned int namelen, unsigned int flags, unsigned int major, unsigned int  
minor, umode_t mode, uid_t uid, gid_t gid, 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.

namelen

The number of characters in *name*, not including a `NULL` terminator. If this is 0, then *name* must be `NULL`-terminated and the length is computed internally.

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.

uid

The default UID of the file.

gid

– undescrbed –

*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

– undescribed –

de

A handle previously created by `devfs_register` or returned from `devfs_find_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
namelen, unsigned int flags, const char * link, unsigned int linklength,
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.

namelen

The number of characters in *name*, not including a `NULL` terminator. If this is 0, then *name* must be `NULL`-terminated and the length is computed internally.

flags

A set of bitwise-ORed flags (DEVFS_FL_*).

link

The destination name.

linklength

The number of characters in *link*, not including a NULL terminator. If this is 0, then *link* must be NULL-terminated and the length is computed internally.

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, unsigned
int namelen, 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.

namelen

The number of characters in *name*, not including a `NULL` terminator. If this is 0, then *name* must be `NULL`-terminated and the length is computed internally.

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_find_handle

Name `devfs_find_handle` — Find the handle of a devfs entry.

Synopsis

```
devfs_handle_t devfs_find_handle (devfs_handle_t dir, const char * name,
unsigned int namelen, unsigned int major, unsigned int minor, char type, int


```

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.

namelen

The number of characters in *name*, not including a `NULL` terminator. If this is 0, then *name* must be `NULL`-terminated and the length is computed internally.

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`. 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.

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

– undescrbed –

size

– undescrbed –

de

The handle to the device entry.

size

The new file size.

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

– undescribed –

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_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

– undescrbed –

name

– undescrbed –

major

The major number for the driver.

name

The name of the driver (as seen in `/proc/devices`).

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 7. 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`.

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.

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.

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.

Chapter 8. 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 9. Video4Linux

video_register_device

Name video_register_device — register video4linux devices

Synopsis

```
int video_register_device (struct video_device * vfd, int type);
```

Arguments

vfd

video device structure we want to register

type

type of device to register

FIXME

needs a semaphore on 2.3.x

The registration code assigns minor numbers based on the type requested. -ENFILE is returned in all the device slots for this category are full. If not then the minor field is set and the driver initialize function is called (if non NULL).

Zero is returned on success.

Valid types are

VFL_TYPE_GRABBER - A frame grabber

VFL_TYPE_VTX - A teletext device

VFL_TYPE_VBI - Vertical blank data (undecoded)

VFL_TYPE_RADIO - A radio card

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 10. 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 success. 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 11. 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 necessary 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 12. 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.

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 isnt 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.

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.

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.

