

# 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. Data Types .....	1
Doubly Linked Lists .....	1
2. Basic C Library Functions .....	44
String Conversions .....	44
String Manipulation .....	66
Bit Operations .....	101
3. Basic Kernel Library Functions .....	119
Bitmap Operations .....	119
Command-line Parsing .....	138
CRC Functions .....	141
idr/ida Functions .....	146
4. Memory Management in Linux .....	163
The Slab Cache .....	163
User Space Memory Access .....	181
More Memory Management Functions .....	188
5. Kernel IPC facilities .....	287
IPC utilities .....	287
6. FIFO Buffer .....	311
kfifo interface .....	311
7. relay interface support .....	345
relay interface .....	345
8. Module Support .....	372
Module Loading .....	372
Inter Module support .....	376
9. Hardware Interfaces .....	377
Interrupt Handling .....	377
DMA Channels .....	396
Resources Management .....	398
MTRR Handling .....	415
PCI Support Library .....	416
PCI Hotplug Support Library .....	552
10. Firmware Interfaces .....	556
DMI Interfaces .....	556
EDD Interfaces .....	564
11. Security Framework .....	570
security_init .....	571
security_module_enable .....	572
securityfs_create_file .....	573
securityfs_create_dir .....	574
securityfs_remove .....	575
12. Audit Interfaces .....	576
audit_log_start .....	577
audit_log_format .....	578
audit_log_end .....	579
audit_log .....	580
audit_log_secctx .....	581
audit_alloc .....	582
__audit_free .....	583
__audit_syscall_entry .....	584
__audit_syscall_exit .....	585
__audit_reusename .....	586

__audit_getname .....	587
__audit_inode .....	588
auditsc_get_stamp .....	589
audit_set_loginuid .....	590
__audit_mq_open .....	591
__audit_mq_sendrecv .....	592
__audit_mq_notify .....	593
__audit_mq_getsetattr .....	594
__audit_ipc_obj .....	595
__audit_ipc_set_perm .....	596
__audit_socketcall .....	597
__audit_fd_pair .....	598
__audit_sockaddr .....	599
__audit_signal_info .....	600
__audit_log_bprm_fcaps .....	601
__audit_log_capset .....	602
audit_core_dumps .....	603
audit_rule_change .....	604
audit_list_rules_send .....	605
parent_len .....	606
audit_compare_dname_path .....	607
13. Accounting Framework .....	608
sys_acct .....	609
acct_collect .....	610
acct_process .....	611
14. Block Devices .....	612
blk_get_backing_dev_info .....	613
blk_delay_queue .....	614
blk_start_queue_async .....	615
blk_start_queue .....	616
blk_stop_queue .....	617
blk_sync_queue .....	618
__blk_run_queue_uncond .....	619
__blk_run_queue .....	620
blk_run_queue_async .....	621
blk_run_queue .....	622
blk_queue_bypass_start .....	623
blk_queue_bypass_end .....	624
blk_cleanup_queue .....	625
blk_init_queue .....	626
blk_make_request .....	627
blk_rq_set_block_pc .....	628
blk_requeue_request .....	629
part_round_stats .....	630
blk_add_request_payload .....	631
generic_make_request .....	632
submit_bio .....	633
blk_insert_cloned_request .....	634
blk_rq_err_bytes .....	635
blk_peek_request .....	636
blk_start_request .....	637
blk_fetch_request .....	638
blk_update_request .....	639
blk_unprep_request .....	640

blk_end_request .....	641
blk_end_request_all .....	642
blk_end_request_cur .....	643
blk_end_request_err .....	644
__blk_end_request .....	645
__blk_end_request_all .....	646
__blk_end_request_cur .....	647
__blk_end_request_err .....	648
rq_flush_dcache_pages .....	649
blk_lld_busy .....	650
blk_rq_unprep_clone .....	651
blk_rq_prep_clone .....	652
blk_start_plug .....	653
blk_pm_runtime_init .....	654
blk_pre_runtime_suspend .....	655
blk_post_runtime_suspend .....	656
blk_pre_runtime_resume .....	657
blk_post_runtime_resume .....	658
__blk_drain_queue .....	659
rq_ioc .....	660
__get_request .....	661
get_request .....	662
blk_attempt_plug_merge .....	663
blk_cloned_rq_check_limits .....	664
blk_end_bidi_request .....	665
__blk_end_bidi_request .....	666
blk_rq_map_user_iov .....	667
blk_rq_unmap_user .....	668
blk_rq_map_kern .....	669
blk_release_queue .....	670
blk_queue_prep_rq .....	671
blk_queue_unprep_rq .....	672
blk_set_default_limits .....	673
blk_set_stacking_limits .....	674
blk_queue_make_request .....	675
blk_queue_bounce_limit .....	676
blk_queue_max_hw_sectors .....	677
blk_queue_chunk_sectors .....	678
blk_queue_max_discard_sectors .....	679
blk_queue_max_write_same_sectors .....	680
blk_queue_max_segments .....	681
blk_queue_max_segment_size .....	682
blk_queue_logical_block_size .....	683
blk_queue_physical_block_size .....	684
blk_queue_alignment_offset .....	685
blk_limits_io_min .....	686
blk_queue_io_min .....	687
blk_limits_io_opt .....	688
blk_queue_io_opt .....	689
blk_queue_stack_limits .....	690
blk_stack_limits .....	691
bdev_stack_limits .....	692
disk_stack_limits .....	693
blk_queue_dma_pad .....	694

blk_queue_update_dma_pad .....	695
blk_queue_dma_drain .....	696
blk_queue_segment_boundary .....	697
blk_queue_virt_boundary .....	698
blk_queue_dma_alignment .....	699
blk_queue_update_dma_alignment .....	700
blk_queue_flush .....	701
blk_execute_rq_nowait .....	702
blk_execute_rq .....	703
blkdev_issue_flush .....	704
blkdev_issue_discard .....	705
blkdev_issue_write_same .....	706
blkdev_issue_zeroout .....	707
blk_queue_find_tag .....	708
blk_free_tags .....	709
blk_queue_free_tags .....	710
blk_init_tags .....	711
blk_queue_init_tags .....	712
blk_queue_resize_tags .....	713
blk_queue_end_tag .....	714
blk_queue_start_tag .....	715
blk_queue_invalidate_tags .....	716
__blk_queue_free_tags .....	717
blk_rq_count_integrity_sg .....	718
blk_rq_map_integrity_sg .....	719
blk_integrity_compare .....	720
blk_integrity_register .....	721
blk_integrity_unregister .....	722
blk_trace_ioctl .....	723
blk_trace_shutdown .....	724
blk_add_trace_rq .....	725
blk_add_trace_bio .....	726
blk_add_trace_bio_remap .....	727
blk_add_trace_rq_remap .....	728
blk_mangle_minor .....	729
blk_alloc_devt .....	730
blk_free_devt .....	731
disk_replace_part_tbl .....	732
disk_expand_part_tbl .....	733
disk_block_events .....	734
disk_unblock_events .....	735
disk_flush_events .....	736
disk_clear_events .....	737
disk_get_part .....	738
disk_part_iter_init .....	739
disk_part_iter_next .....	740
disk_part_iter_exit .....	741
disk_map_sector_rcu .....	742
register_blkdev .....	743
add_disk .....	744
get_gendisk .....	745
bdget_disk .....	746
15. Char devices .....	747
register_chrdev_region .....	748

alloc_chrdev_region .....	749
__register_chrdev .....	750
unregister_chrdev_region .....	751
__unregister_chrdev .....	752
cdev_add .....	753
cdev_del .....	754
cdev_alloc .....	755
cdev_init .....	756
16. Miscellaneous Devices .....	757
misc_register .....	758
misc_deregister .....	759
17. Clock Framework .....	760
struct clk_notifier .....	761
struct clk_notifier_data .....	762
clk_notifier_register .....	763
clk_notifier_unregister .....	764
clk_get_accuracy .....	765
clk_set_phase .....	766
clk_get_phase .....	767
clk_is_match .....	768
clk_prepare .....	769
clk_unprepare .....	770
clk_get .....	771
devm_clk_get .....	772
clk_enable .....	773
clk_disable .....	774
clk_get_rate .....	775
clk_put .....	776
devm_clk_put .....	777
clk_round_rate .....	778
clk_set_rate .....	779
clk_has_parent .....	780
clk_set_rate_range .....	781
clk_set_min_rate .....	782
clk_set_max_rate .....	783
clk_set_parent .....	784
clk_get_parent .....	785
clk_get_sys .....	786

---

# **Chapter 1. Data Types**

## **Doubly Linked Lists**



## Name

`list_add` — add a new entry

## Synopsis

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

## Arguments

*new*    new entry to be added

*head*   list head to add it after

## Description

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

## Name

`list_add_tail` — add a new entry

## Synopsis

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

## Arguments

*new*    new entry to be added

*head*   list head to add it before

## Description

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

## Name

`__list_del_entry` — deletes entry from list.

## Synopsis

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

## Arguments

*entry* the element to delete from the list.

## Note

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

## Name

`list_replace` — replace old entry by new one

## Synopsis

```
void list_replace (struct list_head * old, struct list_head * new);
```

## Arguments

*old* the element to be replaced

*new* the new element to insert

## Description

If *old* was empty, it will be overwritten.

## Name

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

## Synopsis

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

## Arguments

*entry* the element to delete from the list.

## Name

`list_move` — delete from one list and add as another's head

## Synopsis

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

## Arguments

*list* the entry to move

*head* the head that will precede our entry

## Name

`list_move_tail` — delete from one list and add as another's tail

## Synopsis

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

## Arguments

*list* the entry to move

*head* the head that will follow our entry

## Name

`list_is_last` — tests whether *list* is the last entry in list *head*

## Synopsis

```
int list_is_last (const struct list_head * list, const struct list_head  
* head);
```

## Arguments

*list* the entry to test

*head* the head of the list



## Name

`list_empty` — tests whether a list is empty

## Synopsis

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

## Arguments

*head* the list to test.

## Name

`list_empty_careful` — tests whether a list is empty and not being modified

## Synopsis

```
int list_empty_careful (const struct list_head * head);
```

## Arguments

*head* the list to test

## Description

tests whether a list is empty `_and_` checks that no other CPU might be in the process of modifying either member (next or prev)

## NOTE

using `list_empty_careful` without synchronization can only be safe if the only activity that can happen to the list entry is `list_del_init`. Eg. it cannot be used if another CPU could re-`list_add` it.

## Name

`list_rotate_left` — rotate the list to the left

## Synopsis

```
void list_rotate_left (struct list_head * head);
```

## Arguments

*head* the head of the list

## Name

`list_is_singular` — tests whether a list has just one entry.

## Synopsis

```
int list_is_singular (const struct list_head * head);
```

## Arguments

*head* the list to test.

## Name

`list_cut_position` — cut a list into two

## Synopsis

```
void list_cut_position (struct list_head * list, struct list_head *  
head, struct list_head * entry);
```

## Arguments

*list*     a new list to add all removed entries

*head*    a list with entries

*entry*   an entry within head, could be the head itself and if so we won't cut the list

## Description

This helper moves the initial part of *head*, up to and including *entry*, from *head* to *list*. You should pass on *entry* an element you know is on *head*. *list* should be an empty list or a list you do not care about losing its data.

## Name

`list_splice` — join two lists, this is designed for stacks

## Synopsis

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

## Arguments

*list* the new list to add.

*head* the place to add it in the first list.

## Name

`list_splice_tail` — join two lists, each list being a queue

## Synopsis

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

## Arguments

*list* the new list to add.

*head* the place to add it in the first list.

## Name

`list_splice_init` — join two lists and reinitialise the emptied list.

## Synopsis

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

## Arguments

*list* the new list to add.

*head* the place to add it in the first list.

## Description

The list at *list* is reinitialised



## Name

`list_splice_tail_init` — join two lists and reinitialise the emptied list

## Synopsis

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

## Arguments

*list* the new list to add.

*head* the place to add it in the first list.

## Description

Each of the lists is a queue. The list at *list* is reinitialised

## Name

`list_entry` — get the struct for this entry

## Synopsis

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

## Arguments

*ptr*        the struct `list_head` pointer.

*type*       the type of the struct this is embedded in.

*member*    the name of the `list_head` within the struct.

## Name

`list_first_entry` — get the first element from a list

## Synopsis

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

## Arguments

*ptr*        the list head to take the element from.

*type*       the type of the struct this is embedded in.

*member*    the name of the `list_head` within the struct.

## Description

Note, that list is expected to be not empty.

## Name

`list_last_entry` — get the last element from a list

## Synopsis

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

## Arguments

*ptr*        the list head to take the element from.

*type*       the type of the struct this is embedded in.

*member*    the name of the `list_head` within the struct.

## Description

Note, that list is expected to be not empty.

## Name

`list_first_entry_or_null` — get the first element from a list

## Synopsis

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

## Arguments

*ptr*        the list head to take the element from.

*type*       the type of the struct this is embedded in.

*member*    the name of the `list_head` within the struct.

## Description

Note that if the list is empty, it returns `NULL`.

## Name

`list_next_entry` — get the next element in list

## Synopsis

```
list_next_entry ( pos, member );
```

## Arguments

*pos*        the type \* to cursor

*member*    the name of the list\_head within the struct.

## Name

`list_prev_entry` — get the prev element in list

## Synopsis

```
list_prev_entry ( pos, member );
```

## Arguments

*pos*        the type \* to cursor

*member*    the name of the list\_head within the struct.

## Name

`list_for_each` — iterate over a list

## Synopsis

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

## Arguments

*pos*     the struct `list_head` to use as a loop cursor.

*head*   the head for your list.



## Name

`list_for_each_prev` — iterate over a list backwards

## Synopsis

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

## Arguments

*pos*     the struct `list_head` to use as a loop cursor.

*head*   the head for your list.

## Name

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

## Synopsis

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

## Arguments

*pos*     the struct `list_head` to use as a loop cursor.

*n*       another struct `list_head` to use as temporary storage

*head*    the head for your list.

## Name

`list_for_each_prev_safe` — iterate over a list backwards safe against removal of list entry

## Synopsis

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

## Arguments

*pos*     the struct `list_head` to use as a loop cursor.

*n*       another struct `list_head` to use as temporary storage

*head*   the head for your list.

## Name

`list_for_each_entry` — iterate over list of given type

## Synopsis

```
list_for_each_entry ( pos, head, member );
```

## Arguments

*pos*        the type \* to use as a loop cursor.

*head*      the head for your list.

*member*    the name of the list\_head within the struct.

## Name

`list_for_each_entry_reverse` — iterate backwards over list of given type.

## Synopsis

```
list_for_each_entry_reverse ( pos, head, member );
```

## Arguments

*pos*        the type \* to use as a loop cursor.

*head*      the head for your list.

*member*    the name of the list\_head within the struct.

## Name

`list_prepare_entry` — prepare a pos entry for use in `list_for_each_entry_continue`

## Synopsis

```
list_prepare_entry ( pos, head, member );
```

## Arguments

<i>pos</i>	the type * to use as a start point
<i>head</i>	the head of the list
<i>member</i>	the name of the list_head within the struct.

## Description

Prepares a pos entry for use as a start point in `list_for_each_entry_continue`.

## Name

`list_for_each_entry_continue` — continue iteration over list of given type

## Synopsis

```
list_for_each_entry_continue ( pos, head, member );
```

## Arguments

*pos*        the type \* to use as a loop cursor.

*head*      the head for your list.

*member*    the name of the list\_head within the struct.

## Description

Continue to iterate over list of given type, continuing after the current position.

## Name

`list_for_each_entry_continue_reverse` — iterate backwards from the given point

## Synopsis

```
list_for_each_entry_continue_reverse ( pos, head, member );
```

## Arguments

*pos*        the type \* to use as a loop cursor.

*head*      the head for your list.

*member*    the name of the list\_head within the struct.

## Description

Start to iterate over list of given type backwards, continuing after the current position.



## Name

`list_for_each_entry_from` — iterate over list of given type from the current point

## Synopsis

```
list_for_each_entry_from ( pos, head, member );
```

## Arguments

*pos*        the type \* to use as a loop cursor.

*head*      the head for your list.

*member*    the name of the list\_head within the struct.

## Description

Iterate over list of given type, continuing from current position.

## Name

`list_for_each_entry_safe` — iterate over list of given type safe against removal of list entry

## Synopsis

```
list_for_each_entry_safe ( pos, n, head, member );
```

## Arguments

<i>pos</i>	the type * to use as a loop cursor.
<i>n</i>	another type * to use as temporary storage
<i>head</i>	the head for your list.
<i>member</i>	the name of the list_head within the struct.

## Name

`list_for_each_entry_safe_continue` — continue list iteration safe against removal

## Synopsis

```
list_for_each_entry_safe_continue ( pos, n, head, member );
```

## Arguments

*pos*        the type \* to use as a loop cursor.

*n*         another type \* to use as temporary storage

*head*      the head for your list.

*member*   the name of the list\_head within the struct.

## Description

Iterate over list of given type, continuing after current point, safe against removal of list entry.

## Name

`list_for_each_entry_safe_from` — iterate over list from current point safe against removal

## Synopsis

```
list_for_each_entry_safe_from ( pos, n, head, member );
```

## Arguments

*pos*        the type \* to use as a loop cursor.

*n*         another type \* to use as temporary storage

*head*      the head for your list.

*member*   the name of the list\_head within the struct.

## Description

Iterate over list of given type from current point, safe against removal of list entry.

## Name

`list_for_each_entry_safe_reverse` — iterate backwards over list safe against removal

## Synopsis

```
list_for_each_entry_safe_reverse ( pos, n, head, member );
```

## Arguments

*pos*        the type \* to use as a loop cursor.

*n*         another type \* to use as temporary storage

*head*      the head for your list.

*member*    the name of the list\_head within the struct.

## Description

Iterate backwards over list of given type, safe against removal of list entry.

## Name

`list_safe_reset_next` — reset a stale `list_for_each_entry_safe` loop

## Synopsis

```
list_safe_reset_next ( pos, n, member );
```

## Arguments

*pos*        the loop cursor used in the `list_for_each_entry_safe` loop

*n*         temporary storage used in `list_for_each_entry_safe`

*member*    the name of the `list_head` within the struct.

## Description

`list_safe_reset_next` is not safe to use in general if the list may be modified concurrently (eg. the lock is dropped in the loop body). An exception to this is if the cursor element (`pos`) is pinned in the list, and `list_safe_reset_next` is called after re-taking the lock and before completing the current iteration of the loop body.

## Name

`hlist_for_each_entry` — iterate over list of given type

## Synopsis

```
hlist_for_each_entry ( pos, head, member );
```

## Arguments

*pos*        the type \* to use as a loop cursor.

*head*      the head for your list.

*member*    the name of the `hlist_node` within the struct.

## Name

`hlist_for_each_entry_continue` — iterate over a hlist continuing after current point

## Synopsis

```
hlist_for_each_entry_continue ( pos, member );
```

## Arguments

*pos*        the type \* to use as a loop cursor.

*member*    the name of the `hlist_node` within the struct.



## Name

`hlist_for_each_entry_from` — iterate over a hlist continuing from current point

## Synopsis

```
hlist_for_each_entry_from ( pos, member );
```

## Arguments

*pos*        the type \* to use as a loop cursor.

*member*    the name of the `hlist_node` within the struct.

## Name

`hlist_for_each_entry_safe` — iterate over list of given type safe against removal of list entry

## Synopsis

```
hlist_for_each_entry_safe ( pos, n, head, member );
```

## Arguments

<i>pos</i>	the type * to use as a loop cursor.
<i>n</i>	another struct <code>hlist_node</code> to use as temporary storage
<i>head</i>	the head for your list.
<i>member</i>	the name of the <code>hlist_node</code> within the struct.

---

# Chapter 2. Basic C Library Functions

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

## String Conversions

## Name

`simple_strtoul` — convert a string to an unsigned long long

## Synopsis

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

## Arguments

*cp*     The start of the string

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

*base*   The number base to use

## Description

This function is obsolete. Please use `kstrtoul` instead.

## Name

`simple_strtoul` — convert a string to an unsigned long

## Synopsis

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

## Arguments

*cp*     The start of the string

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

*base*   The number base to use

## Description

This function is obsolete. Please use `kstrtoul` instead.

## Name

`simple_strtol` — convert a string to a signed long

## Synopsis

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

## Arguments

*cp*     The start of the string

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

*base*   The number base to use

## Description

This function is obsolete. Please use `kstrtol` instead.

## Name

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

## Synopsis

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

## Arguments

*cp*     The start of the string

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

*base*   The number base to use

## Description

This function is obsolete. Please use `kstrtoll` instead.

## Name

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

## Synopsis

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

## Arguments

*buf*     The buffer to place the result into

*size*    The size of the buffer, including the trailing null space

*fmt*     The format string to use

*args*    Arguments for the format string

## Description

This function generally follows C99 `vsnprintf`, but has some

## extensions and a few limitations

`n` is unsupported `p*` is handled by `pointer`

See `pointer` or `Documentation/printk-formats.txt` for more extensive description.

**\*\* Please update the documentation in both places when making changes \*\***

The return value is the number of characters which would be generated for the given input, excluding the trailing `'\0'`, as per ISO C99. If you want to have the exact number of characters written into *buf* as return value (not including the trailing `'\0'`), use `vscnprintf`. If the return is greater than or equal to *size*, the resulting string is truncated.

If you're not already dealing with a `va_list` consider using `snprintf`.



## Name

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

## Synopsis

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

## Arguments

*buf*     The buffer to place the result into

*size*    The size of the buffer, including the trailing null space

*fmt*     The format string to use

*args*    Arguments for the format string

## Description

The return value is the number of characters which have been written into the *buf* not including the trailing `\0`. If *size* is `== 0` the function returns 0.

If you're not already dealing with a `va_list` consider using `scnprintf`.

See the `vsnprintf` documentation for format string extensions over C99.

## Name

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

## Synopsis

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

## Arguments

*buf*     The buffer to place the result into

*size*    The size of the buffer, including the trailing null space

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

...      variable arguments

## Description

The return value is the number of characters which would be generated for the given input, excluding the trailing null, as per ISO C99. If the return is greater than or equal to *size*, the resulting string is truncated.

See the `vsnprintf` documentation for format string extensions over C99.

## Name

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

## Synopsis

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

## Arguments

*buf*     The buffer to place the result into

*size*    The size of the buffer, including the trailing null space

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

...      variable arguments

## Description

The return value is the number of characters written into *buf* not including the trailing `'\0'`. If *size* is `= 0` the function returns 0.

## Name

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

## Synopsis

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

## Arguments

*buf*     The buffer to place the result into

*fmt*     The format string to use

*args*    Arguments for the format string

## Description

The function returns the number of characters written into *buf*. Use `vsnprintf` or `vsnprintf` in order to avoid buffer overflows.

If you're not already dealing with a `va_list` consider using `sprintf`.

See the `vsnprintf` documentation for format string extensions over C99.

## Name

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

## Synopsis

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

## Arguments

*buf*    The buffer to place the result into

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

...    variable arguments

## Description

The function returns the number of characters written into *buf*. Use `snprintf` or `scnprintf` in order to avoid buffer overflows.

See the `vsnprintf` documentation for format string extensions over C99.

## Name

`vbin_printf` — Parse a format string and place args' binary value in a buffer

## Synopsis

```
int vbin_printf (u32 * bin_buf, size_t size, const char * fmt, va_list  
args);
```

## Arguments

<i>bin_buf</i>	The buffer to place args' binary value
<i>size</i>	The size of the buffer(by words(32bits), not characters)
<i>fmt</i>	The format string to use
<i>args</i>	Arguments for the format string

## Description

The format follows C99 `vsnprintf`, except `n` is ignored, and its argument is skipped.

The return value is the number of words(32bits) which would be generated for the given input.

## NOTE

If the return value is greater than *size*, the resulting *bin\_buf* is NOT valid for `bstr_printf`.

## Name

`bstr_printf` — Format a string from binary arguments and place it in a buffer

## Synopsis

```
int bstr_printf (char * buf, size_t size, const char * fmt, const u32  
* bin_buf);
```

## Arguments

<i>buf</i>	The buffer to place the result into
<i>size</i>	The size of the buffer, including the trailing null space
<i>fmt</i>	The format string to use
<i>bin_buf</i>	Binary arguments for the format string

## Description

This function like C99 `vsnprintf`, but the difference is that `vsnprintf` gets arguments from stack, and `bstr_printf` gets arguments from *bin\_buf* which is a binary buffer that generated by `vbin_printf`.

The format follows C99 `vsnprintf`, but has some extensions: see `vsnprintf` comment for details.

The return value is the number of characters which would be generated for the given input, excluding the trailing `'\0'`, as per ISO C99. If you want to have the exact number of characters written into *buf* as return value (not including the trailing `'\0'`), use `vscnprintf`. If the return is greater than or equal to *size*, the resulting string is truncated.

## Name

bprintf — Parse a format string and place args' binary value in a buffer

## Synopsis

```
int bprintf (u32 * bin_buf, size_t size, const char * fmt, ...);
```

## Arguments

<i>bin_buf</i>	The buffer to place args' binary value
<i>size</i>	The size of the buffer(by words(32bits), not characters)
<i>fmt</i>	The format string to use @...: Arguments for the format string
...	variable arguments

## Description

The function returns the number of words(u32) written into *bin\_buf*.



## Name

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

## Synopsis

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

## Arguments

*buf*    input buffer

*fmt*    format of buffer

*args*   arguments

## Name

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

## Synopsis

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

## Arguments

*buf*    input buffer

*fmt*    formatting of buffer @...: resulting arguments

...    variable arguments

## Name

`kstrtol` — convert a string to a long

## Synopsis

```
int kstrtol (const char * s, unsigned int base, long * res);
```

## Arguments

- s*        The start of the string. The string must be null-terminated, and may also include a single newline before its terminating null. The first character may also be a plus sign or a minus sign.
- base*     The number base to use. The maximum supported base is 16. If base is given as 0, then the base of the string is automatically detected with the conventional semantics - If it begins with 0x the number will be parsed as a hexadecimal (case insensitive), if it otherwise begins with 0, it will be parsed as an octal number. Otherwise it will be parsed as a decimal.
- res*      Where to write the result of the conversion on success.

## Description

Returns 0 on success, -ERANGE on overflow and -EINVAL on parsing error. Used as a replacement for the obsolete `simple_strtoul`. Return code must be checked.

## Name

kstrtoul — convert a string to an unsigned long

## Synopsis

```
int kstrtoul (const char * s, unsigned int base, unsigned long * res);
```

## Arguments

- s*        The start of the string. The string must be null-terminated, and may also include a single newline before its terminating null. The first character may also be a plus sign, but not a minus sign.
- base*     The number base to use. The maximum supported base is 16. If base is given as 0, then the base of the string is automatically detected with the conventional semantics - If it begins with 0x the number will be parsed as a hexadecimal (case insensitive), if it otherwise begins with 0, it will be parsed as an octal number. Otherwise it will be parsed as a decimal.
- res*      Where to write the result of the conversion on success.

## Description

Returns 0 on success, -ERANGE on overflow and -EINVAL on parsing error. Used as a replacement for the obsolete `simple_strtoul`. Return code must be checked.

## Name

kstrtoull — convert a string to an unsigned long long

## Synopsis

```
int kstrtoull (const char * s, unsigned int base, unsigned long long  
* res);
```

## Arguments

- s*        The start of the string. The string must be null-terminated, and may also include a single newline before its terminating null. The first character may also be a plus sign, but not a minus sign.
- base*    The number base to use. The maximum supported base is 16. If base is given as 0, then the base of the string is automatically detected with the conventional semantics - If it begins with 0x the number will be parsed as a hexadecimal (case insensitive), if it otherwise begins with 0, it will be parsed as an octal number. Otherwise it will be parsed as a decimal.
- res*      Where to write the result of the conversion on success.

## Description

Returns 0 on success, -ERANGE on overflow and -EINVAL on parsing error. Used as a replacement for the obsolete `simple_strtoull`. Return code must be checked.

## Name

kstrtoll — convert a string to a long long

## Synopsis

```
int kstrtoll (const char * s, unsigned int base, long long * res);
```

## Arguments

- s*        The start of the string. The string must be null-terminated, and may also include a single newline before its terminating null. The first character may also be a plus sign or a minus sign.
- base*     The number base to use. The maximum supported base is 16. If base is given as 0, then the base of the string is automatically detected with the conventional semantics - If it begins with 0x the number will be parsed as a hexadecimal (case insensitive), if it otherwise begins with 0, it will be parsed as an octal number. Otherwise it will be parsed as a decimal.
- res*      Where to write the result of the conversion on success.

## Description

Returns 0 on success, -ERANGE on overflow and -EINVAL on parsing error. Used as a replacement for the obsolete `simple_strtoll`. Return code must be checked.

## Name

kstrtouint — convert a string to an unsigned int

## Synopsis

```
int kstrtouint (const char * s, unsigned int base, unsigned int * res);
```

## Arguments

*s*        The start of the string. The string must be null-terminated, and may also include a single newline before its terminating null. The first character may also be a plus sign, but not a minus sign.

*base*     The number base to use. The maximum supported base is 16. If base is given as 0, then the base of the string is automatically detected with the conventional semantics - If it begins with 0x the number will be parsed as a hexadecimal (case insensitive), if it otherwise begins with 0, it will be parsed as an octal number. Otherwise it will be parsed as a decimal.

*res*      Where to write the result of the conversion on success.

## Description

Returns 0 on success, -ERANGE on overflow and -EINVAL on parsing error. Used as a replacement for the obsolete `simple_strtoul`. Return code must be checked.

## Name

kstrtoint — convert a string to an int

## Synopsis

```
int kstrtoint (const char * s, unsigned int base, int * res);
```

## Arguments

- |             |   |
|-------------|---|
| <i>s</i>    | The start of the string. The string must be null-terminated, and may also include a single newline before its terminating null. The first character may also be a plus sign or a minus sign.  |
| <i>base</i> | The number base to use. The maximum supported base is 16. If base is given as 0, then the base of the string is automatically detected with the conventional semantics - If it begins with 0x the number will be parsed as a hexadecimal (case insensitive), if it otherwise begins with 0, it will be parsed as an octal number. Otherwise it will be parsed as a decimal. |
| <i>res</i>  | Where to write the result of the conversion on success.   |

## Description

Returns 0 on success, -ERANGE on overflow and -EINVAL on parsing error. Used as a replacement for the obsolete `simple_strtoul`. Return code must be checked.



## Name

kstrtobool — convert common user inputs into boolean values

## Synopsis

```
int kstrtobool (const char * s, bool * res);
```

## Arguments

*s*     input string

*res*   result

## Description

This routine returns 0 iff the first character is one of 'Yy1Nn0', or [oO][NnFf] for “on” and “off”. Otherwise it will return -EINVAL. Value pointed to by *res* is updated upon finding a match.

# String Manipulation

## Name

strncasecmp — Case insensitive, length-limited string comparison

## Synopsis

```
int strncasecmp (const char * s1, const char * s2, size_t len);
```

## Arguments

*s1*    One string

*s2*    The other string

*len*   the maximum number of characters to compare

## Name

`strcpy` — Copy a NUL terminated string

## Synopsis

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

## Arguments

*dest*    Where to copy the string to

*src*     Where to copy the string from

## Name

`strncpy` — Copy a length-limited, C-string

## Synopsis

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

## Arguments

*dest*     Where to copy the string to

*src*     Where to copy the string from

*count*   The maximum number of bytes to copy

## Description

The result is not NUL-terminated if the source exceeds *count* bytes.

In the case where the length of *src* is less than that of *count*, the remainder of *dest* will be padded with NUL.

## Name

`strncpy` — Copy a C-string into a sized buffer

## Synopsis

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

## Arguments

*dest*    Where to copy the string to

*src*     Where to copy the string from

*size*    size of destination buffer

## BSD

the result is always a valid NUL-terminated string that fits in the buffer (unless, of course, the buffer size is zero). It does not pad out the result like `strncpy` does.

## Name

`strncpy` — Copy a C-string into a sized buffer

## Synopsis

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

## Arguments

*dest*     Where to copy the string to

*src*     Where to copy the string from

*count*   Size of destination buffer

## Description

Copy the string, or as much of it as fits, into the *dest* buffer. The routine returns the number of characters copied (not including the trailing NUL) or -E2BIG if the destination buffer wasn't big enough. The behavior is undefined if the string buffers overlap. The destination buffer is always NUL terminated, unless it's zero-sized.

Preferred to `strncpy` since the API doesn't require reading memory from the *src* string beyond the specified “count” bytes, and since the return value is easier to error-check than `strncpy`'s. In addition, the implementation is robust to the string changing out from underneath it, unlike the current `strncpy` implementation.

Preferred to `strncpy` since it always returns a valid string, and doesn't unnecessarily force the tail of the destination buffer to be zeroed. If the zeroing is desired, it's likely cleaner to use `strncpy` with an overflow test, then just `memset` the tail of the *dest* buffer.

## Name

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

## Synopsis

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

## Arguments

*dest*    The string to be appended to

*src*     The string to append to it

## Name

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

## Synopsis

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

## Arguments

*dest*     The string to be appended to

*src*     The string to append to it

*count*   The maximum numbers of bytes to copy

## Description

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



## Name

`strlcat` — Append a length-limited, C-string to another

## Synopsis

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

## Arguments

*dest*    The string to be appended to

*src*     The string to append to it

*count*   The size of the destination buffer.

## Name

strcmp — Compare two strings

## Synopsis

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

## Arguments

*cs*   One string

*ct*   Another string

## Name

`strncmp` — Compare two length-limited strings

## Synopsis

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

## Arguments

<i>cs</i>	One string
<i>ct</i>	Another string
<i>count</i>	The maximum number of bytes to compare

## Name

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

## Synopsis

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

## Arguments

*s*    The string to be searched

*c*    The character to search for

## Name

`strchrnul` — Find and return a character in a string, or end of string

## Synopsis

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

## Arguments

*s*    The string to be searched

*c*    The character to search for

## Description

Returns pointer to first occurrence of 'c' in s. If c is not found, then return a pointer to the null byte at the end of s.

## Name

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

## Synopsis

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

## Arguments

*s*    The string to be searched

*c*    The character to search for

## Name

`strnchr` — Find a character in a length limited string

## Synopsis

```
char * strnchr (const char * s, size_t count, int c);
```

## Arguments

<i>s</i>	The string to be searched
<i>count</i>	The number of characters to be searched
<i>c</i>	The character to search for

## Name

`skip_spaces` — Removes leading whitespace from *str*.

## Synopsis

```
char * skip_spaces (const char * str);
```

## Arguments

*str* The string to be stripped.

## Description

Returns a pointer to the first non-whitespace character in *str*.



## Name

`strim` — Removes leading and trailing whitespace from *s*.

## Synopsis

```
char * strim (char * s);
```

## Arguments

*s*    The string to be stripped.

## Description

Note that the first trailing whitespace is replaced with a NUL-terminator in the given string *s*. Returns a pointer to the first non-whitespace character in *s*.

## Name

`strlen` — Find the length of a string

## Synopsis

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

## Arguments

*s*    The string to be sized

## Name

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

## Synopsis

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

## Arguments

*s*            The string to be sized

*count*       The maximum number of bytes to search

## Name

`strspn` — Calculate the length of the initial substring of *s* which only contain letters in *accept*

## Synopsis

```
size_t strspn (const char * s, const char * accept);
```

## Arguments

*s*            The string to be searched

*accept*      The string to search for

## Name

`strcspn` — Calculate the length of the initial substring of *s* which does not contain letters in *reject*

## Synopsis

```
size_t strcspn (const char * s, const char * reject);
```

## Arguments

*s*            The string to be searched

*reject*      The string to avoid

## Name

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

## Synopsis

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

## Arguments

*cs*    The string to be searched

*ct*    The characters to search for

## Name

`strsep` — Split a string into tokens

## Synopsis

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

## Arguments

*s*     The string to be searched

*ct*    The characters to search for

## Description

`strsep` updates *s* to point after the token, ready for the next call.

It returns empty tokens, too, behaving exactly like the libc function of that name. In fact, it was stolen from glibc2 and de-fancy-fied. Same semantics, slimmer shape. ;)

## Name

`sysfs_streq` — return true if strings are equal, modulo trailing newline

## Synopsis

```
bool sysfs_streq (const char * s1, const char * s2);
```

## Arguments

*s1*   one string

*s2*   another string

## Description

This routine returns true iff two strings are equal, treating both NUL and newline-then-NUL as equivalent string terminations. It's geared for use with sysfs input strings, which generally terminate with newlines but are compared against values without newlines.



## Name

`memset` — Fill a region of memory with the given value

## Synopsis

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

## Arguments

*s*            Pointer to the start of the area.

*c*            The byte to fill the area with

*count*       The size of the area.

## Description

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

## Name

`memzero_explicit` — Fill a region of memory (e.g. sensitive keying data) with 0s.

## Synopsis

```
void memzero_explicit (void * s, size_t count);
```

## Arguments

*s*            Pointer to the start of the area.

*count*       The size of the area.

## Note

usually using `memset` is just fine (!), but in cases where clearing out `_local_` data at the end of a scope is necessary, `memzero_explicit` should be used instead in order to prevent the compiler from optimising away zeroing.

`memzero_explicit` doesn't need an arch-specific version as it just invokes the one of `memset` implicitly.

## Name

`memcpy` — Copy one area of memory to another

## Synopsis

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

## Arguments

*dest*    Where to copy to

*src*     Where to copy from

*count*   The size of the area.

## Description

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

## Name

`memmove` — Copy one area of memory to another

## Synopsis

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

## Arguments

*dest*    Where to copy to

*src*     Where to copy from

*count*   The size of the area.

## Description

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

## Name

`memcmp` — Compare two areas of memory

## Synopsis

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

## Arguments

*cs*        One area of memory

*ct*        Another area of memory

*count*    The size of the area.

## Name

`bcmp` — returns 0 if and only if the buffers have identical contents.

## Synopsis

```
int bcmp (const void * a, const void * b, size_t len);
```

## Arguments

*a*     pointer to first buffer.

*b*     pointer to second buffer.

*len*   size of buffers.

## Description

The sign or magnitude of a non-zero return value has no particular meaning, and architectures may implement their own more efficient `bcmp`. So while this particular implementation is a simple (tail) call to `memcmp`, do not rely on anything but whether the return value is zero or non-zero.

## Name

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

## Synopsis

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

## Arguments

*addr*    The memory area

*c*       The byte to search for

*size*    The size of the area.

## Description

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

## Name

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

## Synopsis

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

## Arguments

*s1*    The string to be searched

*s2*    The string to search for



## Name

`strnstr` — Find the first substring in a length-limited string

## Synopsis

```
char * strnstr (const char * s1, const char * s2, size_t len);
```

## Arguments

*s1*    The string to be searched

*s2*    The string to search for

*len*   the maximum number of characters to search

## Name

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

## Synopsis

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

## Arguments

*s*    The memory area

*c*    The byte to search for

*n*    The size of the area.

## Description

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

## Name

`memchr_inv` — Find an unmatching character in an area of memory.

## Synopsis

```
void * memchr_inv (const void * start, int c, size_t bytes);
```

## Arguments

*start*    The memory area

*c*        Find a character other than *c*

*bytes*    The size of the area.

## Description

returns the address of the first character other than *c*, or `NULL` if the whole buffer contains just *c*.

## Name

`strreplace` — Replace all occurrences of character in string.

## Synopsis

```
char * strreplace (char * s, char old, char new);
```

## Arguments

*s*      The string to operate on.

*old*    The character being replaced.

*new*    The character *old* is replaced with.

## Description

Returns pointer to the nul byte at the end of *s*.

# Bit Operations

## Name

`set_bit` — Atomically set a bit in memory

## Synopsis

```
void set_bit (long nr, volatile unsigned long * addr);
```

## Arguments

*nr*      the bit to set

*addr*    the address to start counting from

## Description

This function is atomic and may not be reordered. See `__set_bit` if you do not require the atomic guarantees.

## Note

there are no guarantees that this function will not be reordered on non x86 architectures, so if you are writing portable code, make sure not to rely on its reordering guarantees.

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

## Name

`__set_bit` — Set a bit in memory

## Synopsis

```
void __set_bit (long nr, volatile unsigned long * addr);
```

## Arguments

*nr*      the bit to set

*addr*    the address to start counting from

## Description

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

## Name

`clear_bit` — Clears a bit in memory

## Synopsis

```
void clear_bit (long nr, volatile unsigned long * addr);
```

## Arguments

*nr*      Bit to clear

*addr*    Address to start counting from

## Description

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

## Name

`__change_bit` — Toggle a bit in memory

## Synopsis

```
void __change_bit (long nr, volatile unsigned long * addr);
```

## Arguments

*nr*      the bit to change

*addr*    the address to start counting from

## Description

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



## Name

`change_bit` — Toggle a bit in memory

## Synopsis

```
void change_bit (long nr, volatile unsigned long * addr);
```

## Arguments

*nr*      Bit to change

*addr*    Address to start counting from

## Description

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

## Name

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

## Synopsis

```
int test_and_set_bit (long nr, volatile unsigned long * addr);
```

## Arguments

*nr*      Bit to set

*addr*    Address to count from

## Description

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

## Name

`test_and_set_bit_lock` — Set a bit and return its old value for lock

## Synopsis

```
int test_and_set_bit_lock (long nr, volatile unsigned long * addr);
```

## Arguments

*nr*      Bit to set

*addr*    Address to count from

## Description

This is the same as `test_and_set_bit` on x86.

## Name

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

## Synopsis

```
int __test_and_set_bit (long nr, volatile unsigned long * addr);
```

## Arguments

*nr*      Bit to set

*addr*    Address to count from

## Description

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

## Name

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

## Synopsis

```
int test_and_clear_bit (long nr, volatile unsigned long * addr);
```

## Arguments

*nr*      Bit to clear

*addr*    Address to count from

## Description

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

## Name

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

## Synopsis

```
int __test_and_clear_bit (long nr, volatile unsigned long * addr);
```

## Arguments

*nr*      Bit to clear

*addr*    Address to count from

## Description

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

## Note

the operation is performed atomically with respect to the local CPU, but not other CPUs. Portable code should not rely on this behaviour. KVM relies on this behaviour on x86 for modifying memory that is also

## **accessed from a hypervisor on the same CPU if running in a VM**

don't change this without also updating arch/x86/kernel/kvm.c

## Name

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

## Synopsis

```
int test_and_change_bit (long nr, volatile unsigned long * addr);
```

## Arguments

*nr*      Bit to change

*addr*    Address to count from

## Description

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

## Name

`test_bit` — Determine whether a bit is set

## Synopsis

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

## Arguments

*nr*      bit number to test

*addr*    Address to start counting from



## Name

`__ffs` — find first set bit in word

## Synopsis

```
unsigned long __ffs (unsigned long word);
```

## Arguments

*word* The word to search

## Description

Undefined if no bit exists, so code should check against 0 first.

## Name

`ffz` — find first zero bit in word

## Synopsis

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

## Arguments

*word* The word to search

## Description

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

## Name

ffs — find first set bit in word

## Synopsis

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

## Arguments

*x* the word to search

## Description

This is defined the same way as the libc and compiler builtin ffs routines, therefore differs in spirit from the other bitops.

ffs(value) returns 0 if value is 0 or the position of the first set bit if value is nonzero. The first (least significant) bit is at position 1.

## Name

fls — find last set bit in word

## Synopsis

```
int fls (int x);
```

## Arguments

*x* the word to search

## Description

This is defined in a similar way as the libc and compiler builtin ffs, but returns the position of the most significant set bit.

fls(value) returns 0 if value is 0 or the position of the last set bit if value is nonzero. The last (most significant) bit is at position 32.

## Name

fls64 — find last set bit in a 64-bit word

## Synopsis

```
int fls64 (__u64 x);
```

## Arguments

*x* the word to search

## Description

This is defined in a similar way as the libc and compiler builtin `ffsll`, but returns the position of the most significant set bit.

`fls64(value)` returns 0 if value is 0 or the position of the last set bit if value is nonzero. The last (most significant) bit is at position 64.

---

# Chapter 3. Basic Kernel Library Functions

The Linux kernel provides more basic utility functions.

## Bitmap Operations

## Name

`__bitmap_shift_right` — logical right shift of the bits in a bitmap

## Synopsis

```
void __bitmap_shift_right (unsigned long * dst, const unsigned long *  
src, unsigned shift, unsigned nbits);
```

## Arguments

*dst*      destination bitmap

*src*      source bitmap

*shift*    shift by this many bits

*nbits*    bitmap size, in bits

## Description

Shifting right (dividing) means moving bits in the MS -> LS bit direction. Zeros are fed into the vacated MS positions and the LS bits shifted off the bottom are lost.

## Name

`__bitmap_shift_left` — logical left shift of the bits in a bitmap

## Synopsis

```
void __bitmap_shift_left (unsigned long * dst, const unsigned long *  
src, unsigned int shift, unsigned int nbits);
```

## Arguments

*dst*      destination bitmap

*src*      source bitmap

*shift*    shift by this many bits

*nbits*    bitmap size, in bits

## Description

Shifting left (multiplying) means moving bits in the LS -> MS direction. Zeros are fed into the vacated LS bit positions and those MS bits shifted off the top are lost.



## Name

`bitmap_find_next_zero_area_off` — find a contiguous aligned zero area

## Synopsis

```
unsigned long bitmap_find_next_zero_area_off (unsigned long * map,  
unsigned long size, unsigned long start, unsigned int nr, unsigned long  
align_mask, unsigned long align_offset);
```

## Arguments

<i>map</i>	The address to base the search on
<i>size</i>	The bitmap size in bits
<i>start</i>	The bitnumber to start searching at
<i>nr</i>	The number of zeroed bits we're looking for
<i>align_mask</i>	Alignment mask for zero area
<i>align_offset</i>	Alignment offset for zero area.

## Description

The *align\_mask* should be one less than a power of 2; the effect is that the bit offset of all zero areas this function finds plus *align\_offset* is multiple of that power of 2.

## Name

`__bitmap_parse` — convert an ASCII hex string into a bitmap.

## Synopsis

```
int __bitmap_parse (const char * buf, unsigned int buflen, int is_user,  
unsigned long * maskp, int nmaskbits);
```

## Arguments

<i>buf</i>	pointer to buffer containing string.
<i>buflen</i>	buffer size in bytes. If string is smaller than this then it must be terminated with a <code>\0</code> .
<i>is_user</i>	location of buffer, 0 indicates kernel space
<i>maskp</i>	pointer to bitmap array that will contain result.
<i>nmaskbits</i>	size of bitmap, in bits.

## Description

Commas group hex digits into chunks. Each chunk defines exactly 32 bits of the resultant bitmask. No chunk may specify a value larger than 32 bits (`-EOVERFLOW`), and if a chunk specifies a smaller value then leading 0-bits are prepended. `-EINVAL` is returned for illegal characters and for grouping errors such as `"1,,5"`, `","44"`, `","` and `""`. Leading and trailing whitespace accepted, but not embedded whitespace.

## Name

`bitmap_parse_user` — convert an ASCII hex string in a user buffer into a bitmap

## Synopsis

```
int bitmap_parse_user (const char __user * ubuf, unsigned int ulen,  
unsigned long * maskp, int nmaskbits);
```

## Arguments

<i>ubuf</i>	pointer to user buffer containing string.
<i>ulen</i>	buffer size in bytes. If string is smaller than this then it must be terminated with a <code>\0</code> .
<i>maskp</i>	pointer to bitmap array that will contain result.
<i>nmaskbits</i>	size of bitmap, in bits.

## Description

Wrapper for `__bitmap_parse`, providing it with user buffer.

We cannot have this as an inline function in `bitmap.h` because it needs `linux/uaccess.h` to get the `access_ok` declaration and this causes cyclic dependencies.

## Name

`bitmap_print_to_pagebuf` — convert bitmap to list or hex format ASCII string

## Synopsis

```
int bitmap_print_to_pagebuf (bool list, char * buf, const unsigned long  
* maskp, int nmaskbits);
```

## Arguments

<i>list</i>	indicates whether the bitmap must be list
<i>buf</i>	page aligned buffer into which string is placed
<i>maskp</i>	pointer to bitmap to convert
<i>nmaskbits</i>	size of bitmap, in bits

## Description

Output format is a comma-separated list of decimal numbers and ranges if list is specified or hex digits grouped into comma-separated sets of 8 digits/set. Returns the number of characters written to *buf*.

It is assumed that *buf* is a pointer into a `PAGE_SIZE` area and that sufficient storage remains at *buf* to accommodate the `bitmap_print_to_pagebuf` output.

## Name

`bitmap_parselist_user` —

## Synopsis

```
int bitmap_parselist_user (const char __user * ubuf, unsigned int ulen,  
unsigned long * maskp, int nmaskbits);
```

## Arguments

<i>ubuf</i>	pointer to user buffer containing string.
<i>ulen</i>	buffer size in bytes. If string is smaller than this then it must be terminated with a <code>\0</code> .
<i>maskp</i>	pointer to bitmap array that will contain result.
<i>nmaskbits</i>	size of bitmap, in bits.

## Description

Wrapper for `bitmap_parselist`, providing it with user buffer.

We cannot have this as an inline function in `bitmap.h` because it needs `linux/uaccess.h` to get the `access_ok` declaration and this causes cyclic dependencies.

## Name

`bitmap_remap` — Apply map defined by a pair of bitmaps to another bitmap

## Synopsis

```
void bitmap_remap (unsigned long * dst, const unsigned long * src, const  
unsigned long * old, const unsigned long * new, unsigned int nbits);
```

## Arguments

<i>dst</i>	remapped result
<i>src</i>	subset to be remapped
<i>old</i>	defines domain of map
<i>new</i>	defines range of map
<i>nbits</i>	number of bits in each of these bitmaps

## Description

Let *old* and *new* define a mapping of bit positions, such that whatever position is held by the *n*-th set bit in *old* is mapped to the *n*-th set bit in *new*. In the more general case, allowing for the possibility that the weight 'w' of *new* is less than the weight of *old*, map the position of the *n*-th set bit in *old* to the position of the *m*-th set bit in *new*, where  $m == n \% w$ .

If either of the *old* and *new* bitmaps are empty, or if *src* and *dst* point to the same location, then this routine copies *src* to *dst*.

The positions of unset bits in *old* are mapped to themselves (the identify map).

Apply the above specified mapping to *src*, placing the result in *dst*, clearing any bits previously set in *dst*.

For example, lets say that *old* has bits 4 through 7 set, and *new* has bits 12 through 15 set. This defines the mapping of bit position 4 to 12, 5 to 13, 6 to 14 and 7 to 15, and of all other bit positions unchanged. So if say *src* comes into this routine with bits 1, 5 and 7 set, then *dst* should leave with bits 1, 13 and 15 set.

## Name

`bitmap_bitremap` — Apply map defined by a pair of bitmaps to a single bit

## Synopsis

```
int bitmap_bitremap (int olddb, const unsigned long * old, const
unsigned long * new, int bits);
```

## Arguments

*olddb*    bit position to be mapped

*old*     defines domain of map

*new*     defines range of map

*bits*    number of bits in each of these bitmaps

## Description

Let *old* and *new* define a mapping of bit positions, such that whatever position is held by the *n*-th set bit in *old* is mapped to the *n*-th set bit in *new*. In the more general case, allowing for the possibility that the weight '*w*' of *new* is less than the weight of *old*, map the position of the *n*-th set bit in *old* to the position of the *m*-th set bit in *new*, where  $m == n \% w$ .

The positions of unset bits in *old* are mapped to themselves (the identify map).

Apply the above specified mapping to bit position *olddb*, returning the new bit position.

For example, lets say that *old* has bits 4 through 7 set, and *new* has bits 12 through 15 set. This defines the mapping of bit position 4 to 12, 5 to 13, 6 to 14 and 7 to 15, and of all other bit positions unchanged. So if say *olddb* is 5, then this routine returns 13.

## Name

`bitmap_onto` — translate one bitmap relative to another

## Synopsis

```
void bitmap_onto (unsigned long * dst, const unsigned long * orig, const
unsigned long * reimap, unsigned int bits);
```

## Arguments

<i>dst</i>	resulting translated bitmap
<i>orig</i>	original untranslated bitmap
<i>reimap</i>	bitmap relative to which translated
<i>bits</i>	number of bits in each of these bitmaps

## Description

Set the *n*-th bit of *dst* iff there exists some *m* such that the *n*-th bit of *reimap* is set, the *m*-th bit of *orig* is set, and the *n*-th bit of *reimap* is also the *m*-th `_set_` bit of *reimap*. (If you understood the previous sentence the first time your read it, you're overqualified for your current job.)

In other words, *orig* is mapped onto (surjectively) *dst*, using the map { <*n*, *m*> | the *n*-th bit of *reimap* is the *m*-th set bit of *reimap* }.

Any set bits in *orig* above bit number *W*, where *W* is the weight of (number of set bits in) *reimap* are mapped nowhere. In particular, if for all bits *m* set in *orig*, *m* >= *W*, then *dst* will end up empty. In situations where the possibility of such an empty result is not desired, one way to avoid it is to use the `bitmap_fold` operator, below, to first fold the *orig* bitmap over itself so that all its set bits *x* are in the range 0 <= *x* < *W*. The `bitmap_fold` operator does this by setting the bit (*m* % *W*) in *dst*, for each bit (*m*) set in *orig*.

Example [1] for `bitmap_onto`: Let's say *reimap* has bits 30-39 set, and *orig* has bits 1, 3, 5, 7, 9 and 11 set. Then on return from this routine, *dst* will have bits 31, 33, 35, 37 and 39 set.

When bit 0 is set in *orig*, it means turn on the bit in *dst* corresponding to whatever is the first bit (if any) that is turned on in *reimap*. Since bit 0 was off in the above example, we leave off that bit (bit 30) in *dst*.

When bit 1 is set in *orig* (as in the above example), it means turn on the bit in *dst* corresponding to whatever is the second bit that is turned on in *reimap*. The second bit in *reimap* that was turned on in the above example was bit 31, so we turned on bit 31 in *dst*.

Similarly, we turned on bits 33, 35, 37 and 39 in *dst*, because they were the 4th, 6th, 8th and 10th set bits set in *reimap*, and the 4th, 6th, 8th and 10th bits of *orig* (i.e. bits 3, 5, 7 and 9) were also set.

When bit 11 is set in *orig*, it means turn on the bit in *dst* corresponding to whatever is the twelfth bit that is turned on in *reimap*. In the above example, there were only ten bits turned on in *reimap* (30..39), so that bit 11 was set in *orig* had no affect on *dst*.

Example [2] for `bitmap_fold + bitmap_onto`: Let's say *reimap* has these ten bits set: 40 41 42 43 45 48 53 61 74 95 (for the curious, that's 40 plus the first ten terms of the Fibonacci sequence.)

Further lets say we use the following code, invoking `bitmap_fold` then `bitmap_onto`, as suggested above to avoid the possibility of an empty *dst* result:



```
unsigned long *tmp; // a temporary bitmap's bits
```

```
bitmap_fold(tmp, orig, bitmap_weight(relmap, bits), bits); bitmap_onto(dst, tmp, relmap, bits);
```

Then this table shows what various values of *dst* would be, for various *orig*'s. I list the zero-based positions of each set bit. The *tmp* column shows the intermediate result, as computed by using `bitmap_fold` to fold the *orig* bitmap modulo ten (the weight of *relmap*).

```
orig tmp dst 0 0 40 1 1 41 9 9 95 10 0 40 (*) 1 3 5 7 1 3 5 7 41 43 48 61 0 1 2 3 4 0 1 2 3 4 40 41
42 43 45 0 9 18 27 0 9 8 7 40 61 74 95 0 10 20 30 0 40 0 11 22 33 0 1 2 3 40 41 42 43 0 12 24 36 0 2
4 6 40 42 45 53 78 102 211 1 2 8 41 42 74 (*)
```

(\*) For these marked lines, if we hadn't first done `bitmap_fold` into *tmp*, then the *dst* result would have been empty.

If either of *orig* or *relmap* is empty (no set bits), then *dst* will be returned empty.

If (as explained above) the only set bits in *orig* are in positions *m* where  $m \geq W$ , (where *W* is the weight of *relmap*) then *dst* will once again be returned empty.

All bits in *dst* not set by the above rule are cleared.

## Name

`bitmap_fold` — fold larger bitmap into smaller, modulo specified size

## Synopsis

```
void bitmap_fold (unsigned long * dst, const unsigned long * orig,  
unsigned int sz, unsigned int nbits);
```

## Arguments

<i>dst</i>	resulting smaller bitmap
<i>orig</i>	original larger bitmap
<i>sz</i>	specified size
<i>nbits</i>	number of bits in each of these bitmaps

## Description

For each bit `oldbit` in *orig*, set bit `oldbit mod sz` in *dst*. Clear all other bits in *dst*. See further the comment and Example [2] for `bitmap_onto` for why and how to use this.

## Name

`bitmap_find_free_region` — find a contiguous aligned mem region

## Synopsis

```
int bitmap_find_free_region (unsigned long * bitmap, unsigned int bits,  
int order);
```

## Arguments

*bitmap*    array of unsigned longs corresponding to the bitmap

*bits*      number of bits in the bitmap

*order*     region size (log base 2 of number of bits) to find

## Description

Find a region of free (zero) bits in a *bitmap* of *bits* bits and allocate them (set them to one). Only consider regions of length a power (*order*) of two, aligned to that power of two, which makes the search algorithm much faster.

Return the bit offset in bitmap of the allocated region, or -errno on failure.

## Name

`bitmap_release_region` — release allocated bitmap region

## Synopsis

```
void bitmap_release_region (unsigned long * bitmap, unsigned int pos,  
int order);
```

## Arguments

*bitmap*    array of unsigned longs corresponding to the bitmap

*pos*        beginning of bit region to release

*order*      region size (log base 2 of number of bits) to release

## Description

This is the complement to `__bitmap_find_free_region` and releases the found region (by clearing it in the bitmap).

No return value.

## Name

`bitmap_allocate_region` — allocate bitmap region

## Synopsis

```
int bitmap_allocate_region (unsigned long * bitmap, unsigned int pos,  
int order);
```

## Arguments

*bitmap*    array of unsigned longs corresponding to the bitmap

*pos*        beginning of bit region to allocate

*order*      region size (log base 2 of number of bits) to allocate

## Description

Allocate (set bits in) a specified region of a bitmap.

Return 0 on success, or `-EBUSY` if specified region wasn't free (not all bits were zero).

## Name

`bitmap_copy_le` — copy a bitmap, putting the bits into little-endian order.

## Synopsis

```
void bitmap_copy_le (unsigned long * dst, const unsigned long * src,  
unsigned int nbits);
```

## Arguments

*dst*      destination buffer

*src*      bitmap to copy

*nbits*    number of bits in the bitmap

## Description

Require `nbits % BITS_PER_LONG == 0`.

## Name

`__bitmap_parselist` — convert list format ASCII string to bitmap

## Synopsis

```
int __bitmap_parselist (const char * buf, unsigned int buflen, int
is_user, unsigned long * maskp, int nmaskbits);
```

## Arguments

<i>buf</i>	read nul-terminated user string from this buffer
<i>buflen</i>	buffer size in bytes. If string is smaller than this then it must be terminated with a \0.
<i>is_user</i>	location of buffer, 0 indicates kernel space
<i>maskp</i>	write resulting mask here
<i>nmaskbits</i>	number of bits in mask to be written

## Description

Input format is a comma-separated list of decimal numbers and ranges. Consecutively set bits are shown as two hyphen-separated decimal numbers, the smallest and largest bit numbers set in the range.

Returns 0 on success, -errno on invalid input strings.

## Error values

-EINVAL: second number in range smaller than first -EINVAL: invalid character in string -ERANGE: bit number specified too large for mask

## Name

`bitmap_pos_to_ord` — find ordinal of set bit at given position in bitmap

## Synopsis

```
int bitmap_pos_to_ord (const unsigned long * buf, unsigned int pos,  
unsigned int nbits);
```

## Arguments

*buf*      pointer to a bitmap

*pos*      a bit position in *buf* ( $0 \leq pos < nbits$ )

*nbits*    number of valid bit positions in *buf*

## Description

Map the bit at position *pos* in *buf* (of length *nbits*) to the ordinal of which set bit it is. If it is not set or if *pos* is not a valid bit position, map to -1.

If for example, just bits 4 through 7 are set in *buf*, then *pos* values 4 through 7 will get mapped to 0 through 3, respectively, and other *pos* values will get mapped to -1. When *pos* value 7 gets mapped to (returns) *ord* value 3 in this example, that means that bit 7 is the 3rd (starting with 0th) set bit in *buf*.

The bit positions 0 through *bits* are valid positions in *buf*.



## Name

`bitmap_ord_to_pos` — find position of n-th set bit in bitmap

## Synopsis

```
unsigned int bitmap_ord_to_pos (const unsigned long * buf, unsigned int  
ord, unsigned int nbits);
```

## Arguments

*buf*      pointer to bitmap

*ord*      ordinal bit position (n-th set bit, n >= 0)

*nbits*    number of valid bit positions in *buf*

## Description

Map the ordinal offset of bit *ord* in *buf* to its position in *buf*. Value of *ord* should be in range  $0 \leq ord < \text{weight}(buf)$ . If  $ord \geq \text{weight}(buf)$ , returns *nbits*.

If for example, just bits 4 through 7 are set in *buf*, then *ord* values 0 through 3 will get mapped to 4 through 7, respectively, and all other *ord* values returns *nbits*. When *ord* value 3 gets mapped to (returns) *pos* value 7 in this example, that means that the 3rd set bit (starting with 0th) is at position 7 in *buf*.

The bit positions 0 through *nbits*-1 are valid positions in *buf*.

## Command-line Parsing

## Name

`get_option` — Parse integer from an option string

## Synopsis

```
int get_option (char ** str, int * pint);
```

## Arguments

*str*    option string

*pint*   (output) integer value parsed from *str*

## Description

Read an int from an option string; if available accept a subsequent comma as well.

## Return values

0 - no int in string 1 - int found, no subsequent comma 2 - int found including a subsequent comma 3 - hyphen found to denote a range

## Name

`get_options` — Parse a string into a list of integers

## Synopsis

```
char * get_options (const char * str, int nints, int * ints);
```

## Arguments

*str*      String to be parsed

*nints*    size of integer array

*ints*     integer array

## Description

This function parses a string containing a comma-separated list of integers, a hyphen-separated range of `_positive_` integers, or a combination of both. The parse halts when the array is full, or when no more numbers can be retrieved from the string.

Return value is the character in the string which caused the parse to end (typically a null terminator, if *str* is completely parseable).

## Name

memparse — parse a string with mem suffixes into a number

## Synopsis

```
unsigned long long memparse (const char * ptr, char ** retptr);
```

## Arguments

*ptr*       Where parse begins

*retptr*   (output) Optional pointer to next char after parse completes

## Description

Parses a string into a number. The number stored at *ptr* is potentially suffixed with K, M, G, T, P, E.

# CRC Functions

## Name

`crc7_be` — update the CRC7 for the data buffer

## Synopsis

```
u8 crc7_be (u8 crc, const u8 * buffer, size_t len);
```

## Arguments

<i>crc</i>	previous CRC7 value
<i>buffer</i>	data pointer
<i>len</i>	number of bytes in the buffer

## Context

any

## Description

Returns the updated CRC7 value. The CRC7 is left-aligned in the byte (the lsb is always 0), as that makes the computation easier, and all callers want it in that form.

## Name

`crc16` — compute the CRC-16 for the data buffer

## Synopsis

```
u16 crc16 (u16 crc, u8 const * buffer, size_t len);
```

## Arguments

*crc*        previous CRC value

*buffer*    data pointer

*len*        number of bytes in the buffer

## Description

Returns the updated CRC value.

## Name

`crc_itu_t` — Compute the CRC-ITU-T for the data buffer

## Synopsis

```
u16 crc_itu_t (u16 crc, const u8 * buffer, size_t len);
```

## Arguments

<i>crc</i>	previous CRC value
<i>buffer</i>	data pointer
<i>len</i>	number of bytes in the buffer

## Description

Returns the updated CRC value

## Name

lib/crc32.c — Document generation inconsistency

## Oops

### Warning

The template for this document tried to insert the structured comment from the file `lib/crc32.c` at this point, but none was found. This dummy section is inserted to allow generation to continue.



## Name

`crc_ccitt` — recompute the CRC for the data buffer

## Synopsis

```
u16 crc_ccitt (u16 crc, u8 const * buffer, size_t len);
```

## Arguments

*crc*        previous CRC value

*buffer*    data pointer

*len*        number of bytes in the buffer

## idr/ida Functions

idr synchronization (stolen from radix-tree.h)

`idr_find` is able to be called locklessly, using RCU. The caller must ensure calls to this function are made within `rcu_read_lock` regions. Other readers (lock-free or otherwise) and modifications may be running concurrently.

It is still required that the caller manage the synchronization and lifetimes of the items. So if RCU lock-free lookups are used, typically this would mean that the items have their own locks, or are amenable to lock-free access; and that the items are freed by RCU (or only freed after having been deleted from the idr tree \*and\* a `synchronize_rcu` grace period).

IDA - IDR based ID allocator

This is id allocator without id -> pointer translation. Memory usage is much lower than full blown idr because each id only occupies a bit. ida uses a custom leaf node which contains `IDA_BITMAP_BITS` slots.

2007-04-25 written by Tejun Heo <htejungmail.com>

## Name

`idr_preload` — preload for `idr_alloc`

## Synopsis

```
void idr_preload (gfp_t gfp_mask);
```

## Arguments

*gfp\_mask* allocation mask to use for preloading

## Description

Preload per-cpu layer buffer for `idr_alloc`. Can only be used from process context and each `idr_preload` invocation should be matched with `idr_preload_end`. Note that preemption is disabled while preloaded.

The first `idr_alloc` in the preloaded section can be treated as if it were invoked with *gfp\_mask* used for preloading. This allows using more permissive allocation masks for idrs protected by spinlocks.

For example, if `idr_alloc` below fails, the failure can be treated as if `idr_alloc` were called with `GFP_KERNEL` rather than `GFP_NOWAIT`.

```
idr_preload(GFP_KERNEL); spin_lock(lock);
```

```
id = idr_alloc(idr, ptr, start, end, GFP_NOWAIT);
```

```
spin_unlock(lock); idr_preload_end; if (id < 0) error;
```

## Name

`idr_alloc` — allocate new idr entry

## Synopsis

```
int idr_alloc (struct idr * idr, void * ptr, int start, int end, gfp_t  
gfp_mask);
```

## Arguments

<i>idr</i>	the (initialized) idr
<i>ptr</i>	pointer to be associated with the new id
<i>start</i>	the minimum id (inclusive)
<i>end</i>	the maximum id (exclusive, $\leq 0$ for max)
<i>gfp_mask</i>	memory allocation flags

## Description

Allocate an id in  $[start, end)$  and associate it with *ptr*. If no ID is available in the specified range, returns -ENOSPC. On memory allocation failure, returns -ENOMEM.

Note that *end* is treated as max when  $\leq 0$ . This is to always allow using  $start + N$  as *end* as long as  $N$  is inside integer range.

The user is responsible for exclusively synchronizing all operations which may modify *idr*. However, read-only accesses such as `idr_find` or iteration can be performed under RCU read lock provided the user destroys *ptr* in RCU-safe way after removal from idr.

## Name

`idr_alloc_cyclic` — allocate new idr entry in a cyclical fashion

## Synopsis

```
int idr_alloc_cyclic (struct idr * idr, void * ptr, int start, int end,  
gfp_t gfp_mask);
```

## Arguments

<i>idr</i>	the (initialized) idr
<i>ptr</i>	pointer to be associated with the new id
<i>start</i>	the minimum id (inclusive)
<i>end</i>	the maximum id (exclusive, $\leq 0$ for max)
<i>gfp_mask</i>	memory allocation flags

## Description

Essentially the same as `idr_alloc`, but prefers to allocate progressively higher ids if it can. If the “cur” counter wraps, then it will start again at the “start” end of the range and allocate one that has already been used.

## Name

`idr_remove` — remove the given id and free its slot

## Synopsis

```
void idr_remove (struct idr * idp, int id);
```

## Arguments

*idp*    idr handle

*id*     unique key

## Name

`idr_destroy` — release all cached layers within an idr tree

## Synopsis

```
void idr_destroy (struct idr * idp);
```

## Arguments

*idp* idr handle

## Description

Free all id mappings and all `idr_layers`. After this function, *idp* is completely unused and can be freed / recycled. The caller is responsible for ensuring that no one else accesses *idp* during or after `idr_destroy`.

A typical clean-up sequence for objects stored in an idr tree will use `idr_for_each` to free all objects, if necessary, then `idr_destroy` to free up the id mappings and cached `idr_layers`.

## Name

`idr_for_each` — iterate through all stored pointers

## Synopsis

```
int idr_for_each (struct idr * idp, int (*fn) (int id, void *p, void  
*data), void * data);
```

## Arguments

*idp*    idr handle

*fn*     function to be called for each pointer

*data*   data passed back to callback function

## Description

Iterate over the pointers registered with the given idr. The callback function will be called for each pointer currently registered, passing the id, the pointer and the data pointer passed to this function. It is not safe to modify the idr tree while in the callback, so functions such as `idr_get_new` and `idr_remove` are not allowed.

We check the return of *fn* each time. If it returns anything other than 0, we break out and return that value.

The caller must serialize `idr_for_each` vs `idr_get_new` and `idr_remove`.

## Name

`idr_get_next` — lookup next object of id to given id.

## Synopsis

```
void * idr_get_next (struct idr * idp, int * nextidp);
```

## Arguments

*idp*           idr handle

*nextidp*   pointer to lookup key

## Description

Returns pointer to registered object with id, which is next number to given id. After being looked up, *\*nextidp* will be updated for the next iteration.

This function can be called under `rcu_read_lock`, given that the leaf pointers lifetimes are correctly managed.



## Name

`idr_replace` — replace pointer for given id

## Synopsis

```
void * idr_replace (struct idr * idp, void * ptr, int id);
```

## Arguments

*idp* idr handle

*ptr* pointer you want associated with the id

*id* lookup key

## Description

Replace the pointer registered with an id and return the old value. A `-ENOENT` return indicates that *id* was not found. A `-EINVAL` return indicates that *id* was not within valid constraints.

The caller must serialize with writers.

## Name

`idr_init` — initialize idr handle

## Synopsis

```
void idr_init (struct idr * idp);
```

## Arguments

*idp* idr handle

## Description

This function is use to set up the handle (*idp*) that you will pass to the rest of the functions.

## Name

`ida_pre_get` — reserve resources for ida allocation

## Synopsis

```
int ida_pre_get (struct ida * ida, gfp_t gfp_mask);
```

## Arguments

*ida*            ida handle

*gfp\_mask*    memory allocation flag

## Description

This function should be called prior to locking and calling the following function. It preallocates enough memory to satisfy the worst possible allocation.

If the system is REALLY out of memory this function returns 0, otherwise 1.

## Name

`ida_get_new_above` — allocate new ID above or equal to a start id

## Synopsis

```
int ida_get_new_above (struct ida * ida, int starting_id, int * p_id);
```

## Arguments

<i>ida</i>	ida handle
<i>starting_id</i>	id to start search at
<i>p_id</i>	pointer to the allocated handle

## Description

Allocate new ID above or equal to *starting\_id*. It should be called with any required locks.

If memory is required, it will return `-EAGAIN`, you should unlock and go back to the `ida_pre_get` call. If the ida is full, it will return `-ENOSPC`.

*p\_id* returns a value in the range *starting\_id* ... 0x7fffffff.

## Name

`ida_remove` — remove the given ID

## Synopsis

```
void ida_remove (struct ida * ida, int id);
```

## Arguments

*ida*    ida handle

*id*     ID to free

## Name

`ida_destroy` — release all cached layers within an ida tree

## Synopsis

```
void ida_destroy (struct ida * ida);
```

## Arguments

*ida*    ida handle

## Name

`ida_simple_get` — get a new id.

## Synopsis

```
int ida_simple_get (struct ida * ida, unsigned int start, unsigned int  
end, gfp_t gfp_mask);
```

## Arguments

<i>ida</i>	the (initialized) ida.
<i>start</i>	the minimum id (inclusive, < 0x8000000)
<i>end</i>	the maximum id (exclusive, < 0x8000000 or 0)
<i>gfp_mask</i>	memory allocation flags

## Description

Allocates an id in the range `start <= id < end`, or returns `-ENOSPC`. On memory allocation failure, returns `-ENOMEM`.

Use `ida_simple_remove` to get rid of an id.

## Name

`ida_simple_remove` — remove an allocated id.

## Synopsis

```
void ida_simple_remove (struct ida * ida, unsigned int id);
```

## Arguments

*ida* the (initialized) ida.

*id* the id returned by `ida_simple_get`.



## Name

`ida_init` — initialize ida handle

## Synopsis

```
void ida_init (struct ida * ida);
```

## Arguments

*ida*    ida handle

## Description

This function is use to set up the handle (*ida*) that you will pass to the rest of the functions.

---

# **Chapter 4. Memory Management in Linux**

## **The Slab Cache**

## Name

`kmalloc` — allocate memory

## Synopsis

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

## Arguments

*size*     how many bytes of memory are required.

*flags*    the type of memory to allocate.

## Description

`kmalloc` is the normal method of allocating memory for objects smaller than page size in the kernel.

The *flags* argument may be one of:

`GFP_USER` - Allocate memory on behalf of user. May sleep.

`GFP_KERNEL` - Allocate normal kernel ram. May sleep.

`GFP_ATOMIC` - Allocation will not sleep. May use emergency pools. For example, use this inside interrupt handlers.

`GFP_HIGHUSER` - Allocate pages from high memory.

`GFP_NOIO` - Do not do any I/O at all while trying to get memory.

`GFP_NOFS` - Do not make any fs calls while trying to get memory.

`GFP_NOWAIT` - Allocation will not sleep.

`__GFP_THISNODE` - Allocate node-local memory only.

`GFP_DMA` - Allocation suitable for DMA. Should only be used for `kmalloc` caches. Otherwise, use a slab created with `SLAB_DMA`.

Also it is possible to set different flags by OR'ing in one or more of the following additional *flags*:

`__GFP_COLD` - Request cache-cold pages instead of trying to return cache-warm pages.

`__GFP_HIGH` - This allocation has high priority and may use emergency pools.

`__GFP_NOFAIL` - Indicate that this allocation is in no way allowed to fail (think twice before using).

`__GFP_NORETRY` - If memory is not immediately available, then give up at once.

`__GFP_NOWARN` - If allocation fails, don't issue any warnings.

`__GFP_REPEAT` - If allocation fails initially, try once more before failing.

There are other flags available as well, but these are not intended for general use, and so are not documented here. For a full list of potential flags, always refer to `linux/gfp.h`.

## Name

`kmalloc_array` — allocate memory for an array.

## Synopsis

```
void * kmalloc_array (size_t n, size_t size, gfp_t flags);
```

## Arguments

*n*            number of elements.

*size*        element size.

*flags*       the type of memory to allocate (see `kmalloc`).

## Name

`kcalloc` — allocate memory for an array. The memory is set to zero.

## Synopsis

```
void * kcalloc (size_t n, size_t size, gfp_t flags);
```

## Arguments

*n*            number of elements.

*size*        element size.

*flags*       the type of memory to allocate (see `kmalloc`).

## Name

`kzalloc` — allocate memory. The memory is set to zero.

## Synopsis

```
void * kzalloc (size_t size, gfp_t flags);
```

## Arguments

*size*    how many bytes of memory are required.

*flags*   the type of memory to allocate (see `kmalloc`).

## Name

`kzalloc_node` — allocate zeroed memory from a particular memory node.

## Synopsis

```
void * kzalloc_node (size_t size, gfp_t flags, int node);
```

## Arguments

*size*    how many bytes of memory are required.

*flags*   the type of memory to allocate (see `kmalloc`).

*node*    memory node from which to allocate

## Name

`kmem_cache_alloc` — Allocate an object

## Synopsis

```
void * kmem_cache_alloc (struct kmem_cache * cachep, gfp_t flags);
```

## Arguments

*cachep*    The cache to allocate from.

*flags*     See `kmalloc`.

## Description

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



## Name

`kmem_cache_alloc_node` — Allocate an object on the specified node

## Synopsis

```
void * kmem_cache_alloc_node (struct kmem_cache * cachep, gfp_t flags,  
int nodeid);
```

## Arguments

*cachep*    The cache to allocate from.

*flags*    See `kmalloc`.

*nodeid*   node number of the target node.

## Description

Identical to `kmem_cache_alloc` but it will allocate memory on the given node, which can improve the performance for cpu bound structures.

Fallback to other node is possible if `__GFP_THISNODE` is not set.

## Name

`kmem_cache_free` — Deallocate an object

## Synopsis

```
void kmem_cache_free (struct kmem_cache * cachep, void * objp);
```

## Arguments

*cachep*    The cache the allocation was from.

*objp*      The previously allocated object.

## Description

Free an object which was previously allocated from this cache.

## Name

kfree — free previously allocated memory

## Synopsis

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

## Arguments

*objp* pointer returned by kmalloc.

## Description

If *objp* is NULL, no operation is performed.

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

## Name

`ksize` — get the actual amount of memory allocated for a given object

## Synopsis

```
size_t ksize (const void * objp);
```

## Arguments

*objp* Pointer to the object

## Description

`kmalloc` may internally round up allocations and return more memory than requested. `ksize` can be used to determine the actual amount of memory allocated. The caller may use this additional memory, even though a smaller amount of memory was initially specified with the `kmalloc` call. The caller must guarantee that *objp* points to a valid object previously allocated with either `kmalloc` or `kmem_cache_alloc`. The object must not be freed during the duration of the call.

## Name

kfree\_const — conditionally free memory

## Synopsis

```
void kfree_const (const void * x);
```

## Arguments

*x* pointer to the memory

## Description

Function calls kfree only if *x* is not in .rodata section.

## Name

`kstrdup` — allocate space for and copy an existing string

## Synopsis

```
char * kstrdup (const char * s, gfp_t gfp);
```

## Arguments

*s*      the string to duplicate

*gfp*    the GFP mask used in the `kmalloc` call when allocating memory

## Name

`kstrdup_const` — conditionally duplicate an existing const string

## Synopsis

```
const char * kstrdup_const (const char * s, gfp_t gfp);
```

## Arguments

*s*      the string to duplicate

*gfp*    the GFP mask used in the `kmalloc` call when allocating memory

## Description

Function returns source string if it is in `.rodata` section otherwise it fallbacks to `kstrdup`. Strings allocated by `kstrdup_const` should be freed by `kfree_const`.

## Name

`kstrndup` — allocate space for and copy an existing string

## Synopsis

```
char * kstrndup (const char * s, size_t max, gfp_t gfp);
```

## Arguments

*s*      the string to duplicate

*max*    read at most *max* chars from *s*

*gfp*    the GFP mask used in the `kmalloc` call when allocating memory

## Note

Use `kmemdup_nul` instead if the size is known exactly.



## Name

`kmemdup` — duplicate region of memory

## Synopsis

```
void * kmemdup (const void * src, size_t len, gfp_t gfp);
```

## Arguments

*src*    memory region to duplicate

*len*    memory region length

*gfp*    GFP mask to use

## Name

`kmemdup_nul` — Create a NUL-terminated string from unterminated data

## Synopsis

```
char * kmemdup_nul (const char * s, size_t len, gfp_t gfp);
```

## Arguments

*s*      The data to stringify

*len*    The size of the data

*gfp*    the GFP mask used in the `kmalloc` call when allocating memory

## Name

`memdup_user` — duplicate memory region from user space

## Synopsis

```
void * memdup_user (const void __user * src, size_t len);
```

## Arguments

*src* source address in user space

*len* number of bytes to copy

## Description

Returns an `ERR_PTR` on failure.

## Name

`get_user_pages_fast` — pin user pages in memory

## Synopsis

```
int get_user_pages_fast (unsigned long start, int nr_pages, int write,  
struct page ** pages);
```

## Arguments

<i>start</i>	starting user address
<i>nr_pages</i>	number of pages from start to pin
<i>write</i>	whether pages will be written to
<i>pages</i>	array that receives pointers to the pages pinned. Should be at least <i>nr_pages</i> long.

## Description

Returns number of pages pinned. This may be fewer than the number requested. If *nr\_pages* is 0 or negative, returns 0. If no pages were pinned, returns -errno.

`get_user_pages_fast` provides equivalent functionality to `get_user_pages`, operating on current and `current->mm`, with `force=0` and `vma=NULL`. However unlike `get_user_pages`, it must be called without `mmap_sem` held.

`get_user_pages_fast` may take `mmap_sem` and page table locks, so no assumptions can be made about lack of locking. `get_user_pages_fast` is to be implemented in a way that is advantageous (vs `get_user_pages`) when the user memory area is already faulted in and present in ptes. However if the pages have to be faulted in, it may turn out to be slightly slower so callers need to carefully consider what to use. On many architectures, `get_user_pages_fast` simply falls back to `get_user_pages`.

## User Space Memory Access

## Name

`__copy_to_user_inatomic` — Copy a block of data into user space, with less checking.

## Synopsis

```
unsigned long __copy_to_user_inatomic (void __user * to, const void *  
from, unsigned long n);
```

## Arguments

*to*      Destination address, in user space.

*from*    Source address, in kernel space.

*n*        Number of bytes to copy.

## Context

User context only.

## Description

Copy data from kernel space to user space. Caller must check the specified block with `access_ok` before calling this function. The caller should also make sure he pins the user space address so that we don't result in page fault and sleep.

Here we special-case 1, 2 and 4-byte `copy_to_user` invocations. On a fault we return the initial request size (1, 2 or 4), as `copy_to_user` should do. If a store crosses a page boundary and gets a fault, the x86 will not write anything, so this is accurate.

## Name

`__copy_to_user` — Copy a block of data into user space, with less checking.

## Synopsis

```
unsigned long __copy_to_user (void __user * to, const void * from,
unsigned long n);
```

## Arguments

*to*      Destination address, in user space.

*from*    Source address, in kernel space.

*n*       Number of bytes to copy.

## Context

User context only. This function may sleep if pagefaults are enabled.

## Description

Copy data from kernel space to user space. Caller must check the specified block with `access_ok` before calling this function.

Returns number of bytes that could not be copied. On success, this will be zero.

## Name

`__copy_from_user` — Copy a block of data from user space, with less checking.

## Synopsis

```
unsigned long __copy_from_user (void * to, const void __user * from,
unsigned long n);
```

## Arguments

*to*      Destination address, in kernel space.

*from*    Source address, in user space.

*n*        Number of bytes to copy.

## Context

User context only. This function may sleep if pagefaults are enabled.

## Description

Copy data from user space to kernel space. Caller must check the specified block with `access_ok` before calling this function.

Returns number of bytes that could not be copied. On success, this will be zero.

If some data could not be copied, this function will pad the copied data to the requested size using zero bytes.

An alternate version - `__copy_from_user_inatomic` - may be called from atomic context and will fail rather than sleep. In this case the uncopied bytes will *\*NOT\** be padded with zeros. See `fs/filemap.h` for explanation of why this is needed.

## Name

`clear_user` — Zero a block of memory in user space.

## Synopsis

```
unsigned long clear_user (void __user * to, unsigned long n);
```

## Arguments

*to* Destination address, in user space.

*n* Number of bytes to zero.

## Description

Zero a block of memory in user space.

Returns number of bytes that could not be cleared. On success, this will be zero.



## Name

`__clear_user` — Zero a block of memory in user space, with less checking.

## Synopsis

```
unsigned long __clear_user (void __user * to, unsigned long n);
```

## Arguments

*to* Destination address, in user space.

*n* Number of bytes to zero.

## Description

Zero a block of memory in user space. Caller must check the specified block with `access_ok` before calling this function.

Returns number of bytes that could not be cleared. On success, this will be zero.

## Name

`_copy_to_user` — Copy a block of data into user space.

## Synopsis

```
unsigned long _copy_to_user (void __user * to, const void * from,
unsigned n);
```

## Arguments

*to*      Destination address, in user space.

*from*    Source address, in kernel space.

*n*        Number of bytes to copy.

## Context

User context only. This function may sleep if pagefaults are enabled.

## Description

Copy data from kernel space to user space.

Returns number of bytes that could not be copied. On success, this will be zero.

## Name

`_copy_from_user` — Copy a block of data from user space.

## Synopsis

```
unsigned long _copy_from_user (void * to, const void __user * from,
unsigned n);
```

## Arguments

*to*      Destination address, in kernel space.

*from*    Source address, in user space.

*n*        Number of bytes to copy.

## Context

User context only. This function may sleep if pagefaults are enabled.

## Description

Copy data from user space to kernel space.

Returns number of bytes that could not be copied. On success, this will be zero.

If some data could not be copied, this function will pad the copied data to the requested size using zero bytes.

# More Memory Management Functions

## Name

`read_cache_pages` — populate an address space with some pages & start reads against them

## Synopsis

```
int read_cache_pages (struct address_space * mapping, struct list_head  
* pages, int (*filler) (void *, struct page *), void * data);
```

## Arguments

*mapping*    the `address_space`

*pages*      The address of a `list_head` which contains the target pages. These pages have their `->index` populated and are otherwise uninitialised.

*filler*     callback routine for filling a single page.

*data*       private data for the callback routine.

## Description

Hides the details of the LRU cache etc from the filesystems.

## Name

`page_cache_sync_readahead` — generic file readahead

## Synopsis

```
void page_cache_sync_readahead (struct address_space * mapping, struct
file_ra_state * ra, struct file * filp, pgoff_t offset, unsigned long
req_size);
```

## Arguments

<i>mapping</i>	address_space which holds the pagecache and I/O vectors
<i>ra</i>	file_ra_state which holds the readahead state
<i>filp</i>	passed on to ->readpage and ->readpages
<i>offset</i>	start offset into <i>mapping</i> , in pagecache page-sized units
<i>req_size</i>	hint: total size of the read which the caller is performing in pagecache pages

## Description

`page_cache_sync_readahead` should be called when a cache miss happened: it will submit the read. The readahead logic may decide to piggyback more pages onto the read request if access patterns suggest it will improve performance.

## Name

`page_cache_async_readahead` — file readahead for marked pages

## Synopsis

```
void page_cache_async_readahead (struct address_space * mapping, struct  
file_ra_state * ra, struct file * filp, struct page * page, pgoff_t  
offset, unsigned long req_size);
```

## Arguments

<i>mapping</i>	address_space which holds the pagecache and I/O vectors
<i>ra</i>	file_ra_state which holds the readahead state
<i>filp</i>	passed on to <code>-&gt;readpage</code> and <code>-&gt;readpages</code>
<i>page</i>	the page at <i>offset</i> which has the PG_readahead flag set
<i>offset</i>	start offset into <i>mapping</i> , in pagecache page-sized units
<i>req_size</i>	hint: total size of the read which the caller is performing in pagecache pages

## Description

`page_cache_async_readahead` should be called when a page is used which has the PG\_readahead flag; this is a marker to suggest that the application has used up enough of the readahead window that we should start pulling in more pages.

## Name

`delete_from_page_cache` — delete page from page cache

## Synopsis

```
void delete_from_page_cache (struct page * page);
```

## Arguments

*page* the page which the kernel is trying to remove from page cache

## Description

This must be called only on pages that have been verified to be in the page cache and locked. It will never put the page into the free list, the caller has a reference on the page.

## Name

`filemap_flush` — mostly a non-blocking flush

## Synopsis

```
int filemap_flush (struct address_space * mapping);
```

## Arguments

*mapping* target address\_space

## Description

This is a mostly non-blocking flush. Not suitable for data-integrity purposes - I/O may not be started against all dirty pages.



## Name

`filemap_fdatawait_range` — wait for writeback to complete

## Synopsis

```
int filemap_fdatawait_range (struct address_space * mapping, loff_t
start_byte, loff_t end_byte);
```

## Arguments

<i>mapping</i>	address space structure to wait for
<i>start_byte</i>	offset in bytes where the range starts
<i>end_byte</i>	offset in bytes where the range ends (inclusive)

## Description

Walk the list of under-writeback pages of the given address space in the given range and wait for all of them. Check error status of the address space and return it.

Since the error status of the address space is cleared by this function, callers are responsible for checking the return value and handling and/or reporting the error.

## Name

`filemap_fdatawait` — wait for all under-writeback pages to complete

## Synopsis

```
int filemap_fdatawait (struct address_space * mapping);
```

## Arguments

*mapping* address space structure to wait for

## Description

Walk the list of under-writeback pages of the given address space and wait for all of them. Check error status of the address space and return it.

Since the error status of the address space is cleared by this function, callers are responsible for checking the return value and handling and/or reporting the error.

## Name

`filemap_write_and_wait_range` — write out & wait on a file range

## Synopsis

```
int filemap_write_and_wait_range (struct address_space * mapping, loff_t
    lstart, loff_t lend);
```

## Arguments

*mapping*    the address\_space for the pages

*lstart*     offset in bytes where the range starts

*lend*       offset in bytes where the range ends (inclusive)

## Description

Write out and wait upon file offsets *lstart*->*lend*, inclusive.

Note that ``lend'` is inclusive (describes the last byte to be written) so that this function can be used to write to the very end-of-file (`end = -1`).

## Name

`replace_page_cache_page` — replace a pagecache page with a new one

## Synopsis

```
int replace_page_cache_page (struct page * old, struct page * new, gfp_t  
gfp_mask);
```

## Arguments

<i>old</i>	page to be replaced
<i>new</i>	page to replace with
<i>gfp_mask</i>	allocation mode

## Description

This function replaces a page in the pagecache with a new one. On success it acquires the pagecache reference for the new page and drops it for the old page. Both the old and new pages must be locked. This function does not add the new page to the LRU, the caller must do that.

The remove + add is atomic. The only way this function can fail is memory allocation failure.

## Name

`add_to_page_cache_locked` — add a locked page to the pagecache

## Synopsis

```
int add_to_page_cache_locked (struct page * page, struct address_space  
* mapping, pgoff_t offset, gfp_t gfp_mask);
```

## Arguments

<i>page</i>	page to add
<i>mapping</i>	the page's <code>address_space</code>
<i>offset</i>	page index
<i>gfp_mask</i>	page allocation mode

## Description

This function is used to add a page to the pagecache. It must be locked. This function does not add the page to the LRU. The caller must do that.

## Name

`add_page_wait_queue` — Add an arbitrary waiter to a page's wait queue

## Synopsis

```
void add_page_wait_queue (struct page * page, wait_queue_t * waiter);
```

## Arguments

*page*      Page defining the wait queue of interest

*waiter*    Waiter to add to the queue

## Description

Add an arbitrary *waiter* to the wait queue for the nominated *page*.

## Name

`unlock_page` — unlock a locked page

## Synopsis

```
void unlock_page (struct page * page);
```

## Arguments

*page* the page

## Description

Unlocks the page and wakes up sleepers in `__wait_on_page_locked`. Also wakes sleepers in `wait_on_page_writeback` because the wakeup mechanism between PageLocked pages and PageWriteback pages is shared. But that's OK - sleepers in `wait_on_page_writeback` just go back to sleep.

The `mb` is necessary to enforce ordering between the `clear_bit` and the read of the waitqueue (to avoid SMP races with a parallel `wait_on_page_locked`).

## Name

`end_page_writeback` — end writeback against a page

## Synopsis

```
void end_page_writeback (struct page * page);
```

## Arguments

*page* the page



## Name

`__lock_page` — get a lock on the page, assuming we need to sleep to get it

## Synopsis

```
void __lock_page (struct page * page);
```

## Arguments

*page* the page to lock

## Name

`page_cache_next_hole` — find the next hole (not-present entry)

## Synopsis

```
pgoff_t page_cache_next_hole (struct address_space * mapping, pgoff_t
index, unsigned long max_scan);
```

## Arguments

*mapping*    mapping

*index*      index

*max\_scan*   maximum range to search

## Description

Search the set [*index*, min(*index*+*max\_scan*-1, MAX\_INDEX)] for the lowest indexed hole.

## Returns

the index of the hole if found, otherwise returns an index outside of the set specified (in which case 'return - *index* >= *max\_scan*' will be true). In rare cases of index wrap-around, 0 will be returned.

`page_cache_next_hole` may be called under `rcu_read_lock`. However, like `radix_tree_gang_lookup`, this will not atomically search a snapshot of the tree at a single point in time. For example, if a hole is created at index 5, then subsequently a hole is created at index 10, `page_cache_next_hole` covering both indexes may return 10 if called under `rcu_read_lock`.

## Name

`page_cache_prev_hole` — find the prev hole (not-present entry)

## Synopsis

```
pgoff_t page_cache_prev_hole (struct address_space * mapping, pgoff_t
index, unsigned long max_scan);
```

## Arguments

*mapping*    mapping

*index*      index

*max\_scan*   maximum range to search

## Description

Search backwards in the range `[max(index-max_scan+1, 0), index]` for the first hole.

## Returns

the index of the hole if found, otherwise returns an index outside of the set specified (in which case 'index - return >= max\_scan' will be true). In rare cases of wrap-around, `ULONG_MAX` will be returned.

`page_cache_prev_hole` may be called under `rcu_read_lock`. However, like `radix_tree_gang_lookup`, this will not atomically search a snapshot of the tree at a single point in time. For example, if a hole is created at index 10, then subsequently a hole is created at index 5, `page_cache_prev_hole` covering both indexes may return 5 if called under `rcu_read_lock`.

## Name

`find_get_entry` — find and get a page cache entry

## Synopsis

```
struct page * find_get_entry (struct address_space * mapping, pgoff_t
                                offset);
```

## Arguments

*mapping*    the address\_space to search

*offset*     the page cache index

## Description

Looks up the page cache slot at *mapping* & *offset*. If there is a page cache page, it is returned with an increased refcount.

If the slot holds a shadow entry of a previously evicted page, or a swap entry from shmem/tmpfs, it is returned.

Otherwise, NULL is returned.

## Name

`find_lock_entry` — locate, pin and lock a page cache entry

## Synopsis

```
struct page * find_lock_entry (struct address_space * mapping, pgoff_t  
offset);
```

## Arguments

*mapping*    the address\_space to search

*offset*     the page cache index

## Description

Looks up the page cache slot at *mapping* & *offset*. If there is a page cache page, it is returned locked and with an increased refcount.

If the slot holds a shadow entry of a previously evicted page, or a swap entry from shmem/tmpfs, it is returned.

Otherwise, NULL is returned.

`find_lock_entry` may sleep.

## Name

`pagecache_get_page` — find and get a page reference

## Synopsis

```
struct page * pagecache_get_page (struct address_space * mapping,  
pgoff_t offset, int fgp_flags, gfp_t gfp_mask);
```

## Arguments

*mapping*      the address\_space to search

*offset*        the page index

*fgp\_flags*    PCG flags

*gfp\_mask*     gfp mask to use for the page cache data page allocation

## Description

Looks up the page cache slot at *mapping* & *offset*.

PCG flags modify how the page is returned.

## FGP\_ACCESSED

the page will be marked accessed

## FGP\_LOCK

Page is return locked

## FGP\_CREAT

If page is not present then a new page is allocated using *gfp\_mask* and added to the page cache and the VM's LRU list. The page is returned locked and with an increased refcount. Otherwise, NULL is returned.

If FGP\_LOCK or FGP\_CREAT are specified then the function may sleep even if the GFP flags specified for FGP\_CREAT are atomic.

If there is a page cache page, it is returned with an increased refcount.

## Name

`find_get_pages` — gang pagecache lookup

## Synopsis

```
unsigned find_get_pages (struct address_space * mapping, pgoff_t start,  
unsigned int nr_pages, struct page ** pages);
```

## Arguments

<i>mapping</i>	The <code>address_space</code> to search
<i>start</i>	The starting page index
<i>nr_pages</i>	The maximum number of pages
<i>pages</i>	Where the resulting pages are placed

## Description

`find_get_pages` will search for and return a group of up to *nr\_pages* pages in the mapping. The pages are placed at *pages*. `find_get_pages` takes a reference against the returned pages.

The search returns a group of mapping-contiguous pages with ascending indexes. There may be holes in the indices due to not-present pages.

`find_get_pages` returns the number of pages which were found.

## Name

`find_get_pages_contig` — gang contiguous pagecache lookup

## Synopsis

```
unsigned find_get_pages_contig (struct address_space * mapping, pgoff_t
index, unsigned int nr_pages, struct page ** pages);
```

## Arguments

<i>mapping</i>	The <code>address_space</code> to search
<i>index</i>	The starting page index
<i>nr_pages</i>	The maximum number of pages
<i>pages</i>	Where the resulting pages are placed

## Description

`find_get_pages_contig` works exactly like `find_get_pages`, except that the returned number of pages are guaranteed to be contiguous.

`find_get_pages_contig` returns the number of pages which were found.



## Name

`find_get_pages_tag` — find and return pages that match *tag*

## Synopsis

```
unsigned find_get_pages_tag (struct address_space * mapping, pgoff_t *  
index, int tag, unsigned int nr_pages, struct page ** pages);
```

## Arguments

<i>mapping</i>	the <code>address_space</code> to search
<i>index</i>	the starting page index
<i>tag</i>	the tag index
<i>nr_pages</i>	the maximum number of pages
<i>pages</i>	where the resulting pages are placed

## Description

Like `find_get_pages`, except we only return pages which are tagged with *tag*. We update *index* to index the next page for the traversal.

## Name

`generic_file_read_iter` — generic filesystem read routine

## Synopsis

```
ssize_t generic_file_read_iter (struct kiocb * iocb, struct iov_iter  
* iter);
```

## Arguments

*iocb* kernel I/O control block

*iter* destination for the data read

## Description

This is the “`read_iter`” routine for all filesystems that can use the page cache directly.

## Name

`filemap_fault` — read in file data for page fault handling

## Synopsis

```
int filemap_fault (struct vm_area_struct * vma, struct vm_fault * vmf);
```

## Arguments

*vma*    vma in which the fault was taken

*vmf*    struct `vm_fault` containing details of the fault

## Description

`filemap_fault` is invoked via the vma operations vector for a mapped memory region to read in file data during a page fault.

The goto's are kind of ugly, but this streamlines the normal case of having it in the page cache, and handles the special cases reasonably without having a lot of duplicated code.

`vma->vm_mm->mmap_sem` must be held on entry.

If our return value has `VM_FAULT_RETRY` set, it's because `lock_page_or_retry` returned 0. The `mmap_sem` has usually been released in this case. See `__lock_page_or_retry` for the exception.

If our return value does not have `VM_FAULT_RETRY` set, the `mmap_sem` has not been released.

We never return with `VM_FAULT_RETRY` and a bit from `VM_FAULT_ERROR` set.

## Name

`read_cache_page` — read into page cache, fill it if needed

## Synopsis

```
struct page * read_cache_page (struct address_space * mapping, pgoff_t
index, int (*filler) (void *, struct page *), void * data);
```

## Arguments

*mapping*    the page's address\_space

*index*      the page index

*filler*     function to perform the read

*data*       first arg to filler(data, page) function, often left as NULL

## Description

Read into the page cache. If a page already exists, and `PageUptodate` is not set, try to fill the page and wait for it to become unlocked.

If the page does not get brought uptodate, return `-EIO`.

## Name

`read_cache_page_gfp` — read into page cache, using specified page allocation flags.

## Synopsis

```
struct page * read_cache_page_gfp (struct address_space * mapping,  
pgoff_t index, gfp_t gfp);
```

## Arguments

*mapping*    the page's address\_space

*index*      the page index

*gfp*        the page allocator flags to use if allocating

## Description

This is the same as “`read_mapping_page(mapping, index, NULL)`”, but with any new page allocations done using the specified allocation flags.

If the page does not get brought uptodate, return `-EIO`.

## Name

`__generic_file_write_iter` — write data to a file

## Synopsis

```
ssize_t __generic_file_write_iter (struct kiocb * iocb, struct iov_iter  
* from);
```

## Arguments

*iocb* IO state structure (file, offset, etc.)

*from* iov\_iter with data to write

## Description

This function does all the work needed for actually writing data to a file. It does all basic checks, removes SUID from the file, updates modification times and calls proper subroutines depending on whether we do direct IO or a standard buffered write.

It expects `i_mutex` to be grabbed unless we work on a block device or similar object which does not need locking at all.

This function does *not* take care of syncing data in case of `O_SYNC` write. A caller has to handle it. This is mainly due to the fact that we want to avoid syncing under `i_mutex`.

## Name

`generic_file_write_iter` — write data to a file

## Synopsis

```
ssize_t generic_file_write_iter (struct kiocb * iocb, struct iov_iter  
* from);
```

## Arguments

*iocb* IO state structure

*from* iov\_iter with data to write

## Description

This is a wrapper around `__generic_file_write_iter` to be used by most filesystems. It takes care of syncing the file in case of `O_SYNC` file and acquires `i_mutex` as needed.

## Name

`try_to_release_page` — release old fs-specific metadata on a page

## Synopsis

```
int try_to_release_page (struct page * page, gfp_t gfp_mask);
```

## Arguments

*page*            the page which the kernel is trying to free

*gfp\_mask*       memory allocation flags (and I/O mode)

## Description

The `address_space` is to try to release any data against the page (presumably at `page->private`). If the release was successful, return `1`. Otherwise return zero.

This may also be called if `PG_fscache` is set on a page, indicating that the page is known to the local caching routines.

The *gfp\_mask* argument specifies whether I/O may be performed to release this page (`__GFP_IO`), and whether the call may block (`__GFP_RECLAIM & __GFP_FS`).



## Name

`zap_vma_ptes` — remove ptes mapping the vma

## Synopsis

```
int zap_vma_ptes (struct vm_area_struct * vma, unsigned long address,  
unsigned long size);
```

## Arguments

*vma*            `vm_area_struct` holding ptes to be zapped

*address*       starting address of pages to zap

*size*           number of bytes to zap

## Description

This function only unmaps ptes assigned to `VM_PFNMAP` vmas.

The entire address range must be fully contained within the vma.

Returns 0 if successful.

## Name

`vm_insert_page` — insert single page into user vma

## Synopsis

```
int vm_insert_page (struct vm_area_struct * vma, unsigned long addr,  
struct page * page);
```

## Arguments

*vma*     user vma to map to

*addr*   target user address of this page

*page*   source kernel page

## Description

This allows drivers to insert individual pages they've allocated into a user vma.

The page has to be a nice clean `_individual_` kernel allocation. If you allocate a compound page, you need to have marked it as such (`__GFP_COMP`), or manually just split the page up yourself (see `split_page`).

NOTE! Traditionally this was done with “`remap_pfn_range`” which took an arbitrary page protection parameter. This doesn't allow that. Your vma protection will have to be set up correctly, which means that if you want a shared writable mapping, you'd better ask for a shared writable mapping!

The page does not need to be reserved.

Usually this function is called from `f_op->mmap` handler under `mm->mmap_sem` write-lock, so it can change `vma->vm_flags`. Caller must set `VM_MIXEDMAP` on `vma` if it wants to call this function from other places, for example from page-fault handler.

## Name

`vm_insert_pfn` — insert single pfn into user vma

## Synopsis

```
int vm_insert_pfn (struct vm_area_struct * vma, unsigned long addr,  
unsigned long pfn);
```

## Arguments

*vma*     user vma to map to

*addr*    target user address of this page

*pfn*     source kernel pfn

## Description

Similar to `vm_insert_page`, this allows drivers to insert individual pages they've allocated into a user vma. Same comments apply.

This function should only be called from a `vm_ops->fault` handler, and in that case the handler should return `NULL`.

*vma* cannot be a COW mapping.

As this is called only for pages that do not currently exist, we do not need to flush old virtual caches or the TLB.

## Name

`vm_insert_pfn_prot` — insert single pfn into user vma with specified pgprot

## Synopsis

```
int vm_insert_pfn_prot (struct vm_area_struct * vma, unsigned long addr,  
unsigned long pfn, pgprot_t pgprot);
```

## Arguments

<i>vma</i>	user vma to map to
<i>addr</i>	target user address of this page
<i>pfn</i>	source kernel pfn
<i>pgprot</i>	pgprot flags for the inserted page

## Description

This is exactly like `vm_insert_pfn`, except that it allows drivers to to override `pgprot` on a per-page basis.

This only makes sense for IO mappings, and it makes no sense for cow mappings. In general, using multiple `vm`s is preferable; `vm_insert_pfn_prot` should only be used if using multiple VMAs is impractical.

## Name

`remap_pfn_range` — remap kernel memory to userspace

## Synopsis

```
int remap_pfn_range (struct vm_area_struct * vma, unsigned long addr,  
unsigned long pfn, unsigned long size, pgprot_t prot);
```

## Arguments

*vma*    user vma to map to

*addr*   target user address to start at

*pfn*    physical address of kernel memory

*size*   size of map area

*prot*   page protection flags for this mapping

## Note

this is only safe if the mm semaphore is held when called.

## Name

`vm_iomap_memory` — remap memory to userspace

## Synopsis

```
int vm_iomap_memory (struct vm_area_struct * vma, phys_addr_t start,
unsigned long len);
```

## Arguments

*vma*      user vma to map to

*start*    start of area

*len*      size of area

## Description

This is a simplified `io_remap_pfn_range` for common driver use. The driver just needs to give us the physical memory range to be mapped, we'll figure out the rest from the vma information.

NOTE! Some drivers might want to tweak `vma->vm_page_prot` first to get whatever write-combining details or similar.

## Name

`unmap_mapping_range` — unmap the portion of all mmaps in the specified `address_space` corresponding to the specified page range in the underlying file.

## Synopsis

```
void unmap_mapping_range (struct address_space * mapping, loff_t const holebegin, loff_t const holelen, int even_cows);
```

## Arguments

<i>mapping</i>	the address space containing mmaps to be unmapped.
<i>holebegin</i>	byte in first page to unmap, relative to the start of the underlying file. This will be rounded down to a <code>PAGE_SIZE</code> boundary. Note that this is different from <code>truncate_pagecache</code> , which must keep the partial page. In contrast, we must get rid of partial pages.
<i>holelen</i>	size of prospective hole in bytes. This will be rounded up to a <code>PAGE_SIZE</code> boundary. A <code>holelen</code> of zero truncates to the end of the file.
<i>even_cows</i>	1 when truncating a file, unmap even private COWed pages; but 0 when invalidating pagecache, don't throw away private data.

## Name

`follow_pfn` — look up PFN at a user virtual address

## Synopsis

```
int follow_pfn (struct vm_area_struct * vma, unsigned long address,
unsigned long * pfn);
```

## Arguments

<i>vma</i>	memory mapping
<i>address</i>	user virtual address
<i>pfn</i>	location to store found PFN

## Description

Only IO mappings and raw PFN mappings are allowed.

Returns zero and the pfn at *pfn* on success, -ve otherwise.



## Name

`vm_unmap_aliases` — unmap outstanding lazy aliases in the vmap layer

## Synopsis

```
void vm_unmap_aliases ( void );
```

## Arguments

*void* no arguments

## Description

The vmap/vmalloc layer lazily flushes kernel virtual mappings primarily to amortize TLB flushing overheads. What this means is that any page you have now, may, in a former life, have been mapped into kernel virtual address by the vmap layer and so there might be some CPUs with TLB entries still referencing that page (additional to the regular 1:1 kernel mapping).

`vm_unmap_aliases` flushes all such lazy mappings. After it returns, we can be sure that none of the pages we have control over will have any aliases from the vmap layer.

## Name

`vm_unmap_ram` — unmap linear kernel address space set up by `vm_map_ram`

## Synopsis

```
void vm_unmap_ram (const void * mem, unsigned int count);
```

## Arguments

*mem*      the pointer returned by `vm_map_ram`

*count*    the count passed to that `vm_map_ram` call (cannot unmap partial)

## Name

`vm_map_ram` — map pages linearly into kernel virtual address (vmalloc space)

## Synopsis

```
void * vm_map_ram (struct page ** pages, unsigned int count, int node,  
pgprot_t prot);
```

## Arguments

*pages*    an array of pointers to the pages to be mapped

*count*    number of pages

*node*     prefer to allocate data structures on this node

*prot*     memory protection to use. `PAGE_KERNEL` for regular RAM

## Description

If you use this function for less than `VMAP_MAX_ALLOC` pages, it could be faster than `vmap` so it's good. But if you mix long-life and short-life objects with `vm_map_ram`, it could consume lots of address space through fragmentation (especially on a 32bit machine). You could see failures in the end. Please use this function for short-lived objects.

## Returns

a pointer to the address that has been mapped, or `NULL` on failure

## Name

`unmap_kernel_range_noflush` — unmap kernel VM area

## Synopsis

```
void unmap_kernel_range_noflush (unsigned long addr, unsigned long
size);
```

## Arguments

*addr* start of the VM area to unmap

*size* size of the VM area to unmap

## Description

Unmap `PFN_UP(size)` pages at *addr*. The VM area *addr* and *size* specify should have been allocated using `get_vm_area` and its friends.

## NOTE

This function does NOT do any cache flushing. The caller is responsible for calling `flush_cache_vunmap` on to-be-mapped areas before calling this function and `flush_tlb_kernel_range` after.

## Name

`unmap_kernel_range` — unmap kernel VM area and flush cache and TLB

## Synopsis

```
void unmap_kernel_range (unsigned long addr, unsigned long size);
```

## Arguments

*addr* start of the VM area to unmap

*size* size of the VM area to unmap

## Description

Similar to `unmap_kernel_range_noflush` but flushes vcache before the unmapping and tlb after.

## Name

`vfree` — release memory allocated by `vmalloc`

## Synopsis

```
void vfree (const void * addr);
```

## Arguments

*addr*    memory base address

## Description

Free the virtually continuous memory area starting at *addr*, as obtained from `vmalloc`, `vmalloc_32` or `__vmalloc`. If *addr* is `NULL`, no operation is performed.

Must not be called in NMI context (strictly speaking, only if we don't have `CONFIG_ARCH_HAVE_NMI_SAFE_CMPXCHG`, but making the calling conventions for `vfree` arch-dependent would be a really bad idea)

## NOTE

assumes that the object at *\*addr* has a size  $\geq$  `sizeof(list_node)`

## Name

`vunmap` — release virtual mapping obtained by `vmap`

## Synopsis

```
void vunmap (const void * addr);
```

## Arguments

*addr*    memory base address

## Description

Free the virtually contiguous memory area starting at *addr*, which was created from the page array passed to `vmap`.

Must not be called in interrupt context.

## Name

`vmap` — map an array of pages into virtually contiguous space

## Synopsis

```
void * vmap (struct page ** pages, unsigned int count, unsigned long  
flags, pgprot_t prot);
```

## Arguments

*pages*    array of page pointers

*count*    number of pages to map

*flags*    `vm_area->flags`

*prot*    page protection for the mapping

## Description

Maps *count* pages from *pages* into contiguous kernel virtual space.



## Name

`vmalloc` — allocate virtually contiguous memory

## Synopsis

```
void * vmalloc (unsigned long size);
```

## Arguments

*size*    allocation size Allocate enough pages to cover *size* from the page level allocator and map them into contiguous kernel virtual space.

## Description

For tight control over page level allocator and protection flags use `__vmalloc` instead.

## Name

`vzalloc` — allocate virtually contiguous memory with zero fill

## Synopsis

```
void * vzalloc (unsigned long size);
```

## Arguments

*size* allocation size Allocate enough pages to cover *size* from the page level allocator and map them into contiguous kernel virtual space. The memory allocated is set to zero.

## Description

For tight control over page level allocator and protection flags use `__vmalloc` instead.

## Name

`vmalloc_user` — allocate zeroed virtually contiguous memory for userspace

## Synopsis

```
void * vmalloc_user (unsigned long size);
```

## Arguments

*size*    allocation size

## Description

The resulting memory area is zeroed so it can be mapped to userspace without leaking data.

## Name

`vmalloc_node` — allocate memory on a specific node

## Synopsis

```
void * vmalloc_node (unsigned long size, int node);
```

## Arguments

*size*    allocation size

*node*    numa node

## Description

Allocate enough pages to cover *size* from the page level allocator and map them into contiguous kernel virtual space.

For tight control over page level allocator and protection flags use `__vmalloc` instead.

## Name

`vzalloc_node` — allocate memory on a specific node with zero fill

## Synopsis

```
void * vzalloc_node (unsigned long size, int node);
```

## Arguments

*size*    allocation size

*node*    numa node

## Description

Allocate enough pages to cover *size* from the page level allocator and map them into contiguous kernel virtual space. The memory allocated is set to zero.

For tight control over page level allocator and protection flags use `__vmalloc_node` instead.

## Name

`vmalloc_32` — allocate virtually contiguous memory (32bit addressable)

## Synopsis

```
void * vmalloc_32 (unsigned long size);
```

## Arguments

*size*    allocation size

## Description

Allocate enough 32bit PA addressable pages to cover *size* from the page level allocator and map them into contiguous kernel virtual space.

## Name

`vmalloc_32_user` — allocate zeroed virtually contiguous 32bit memory

## Synopsis

```
void * vmalloc_32_user (unsigned long size);
```

## Arguments

*size*    allocation size

## Description

The resulting memory area is 32bit addressable and zeroed so it can be mapped to userspace without leaking data.

## Name

`remap_vmalloc_range_partial` — map vmalloc pages to userspace

## Synopsis

```
int remap_vmalloc_range_partial (struct vm_area_struct * vma, unsigned
long uaddr, void * kaddr, unsigned long size);
```

## Arguments

*vma*      vma to cover

*uaddr*    target user address to start at

*kaddr*    virtual address of vmalloc kernel memory

*size*     size of map area

## Returns

0 for success, -Exxx on failure

This function checks that *kaddr* is a valid vmalloc'ed area, and that it is big enough to cover the range starting at *uaddr* in *vma*. Will return failure if that criteria isn't met.

Similar to `remap_pfn_range` (see `mm/memory.c`)



## Name

`remap_vmalloc_range` — map vmalloc pages to userspace

## Synopsis

```
int remap_vmalloc_range (struct vm_area_struct * vma, void * addr,
unsigned long pgoff);
```

## Arguments

*vma*      vma to cover (map full range of vma)

*addr*     vmalloc memory

*pgoff*    number of pages into *addr* before first page to map

## Returns

0 for success, -Exxx on failure

This function checks that *addr* is a valid vmalloc'ed area, and that it is big enough to cover the *vma*. Will return failure if that criteria isn't met.

Similar to `remap_pfn_range` (see `mm/memory.c`)

## Name

`alloc_vm_area` — allocate a range of kernel address space

## Synopsis

```
struct vm_struct * alloc_vm_area (size_t size, pte_t ** ptes);
```

## Arguments

*size*    size of the area

*ptes*    returns the PTEs for the address space

## Returns

NULL on failure, `vm_struct` on success

This function reserves a range of kernel address space, and allocates pagetables to map that range. No actual mappings are created.

If *ptes* is non-NULL, pointers to the PTEs (in `init_mm`) allocated for the VM area are returned.

## Name

`alloc_pages_exact_nid` — allocate an exact number of physically-contiguous pages on a node.

## Synopsis

```
void * alloc_pages_exact_nid (int nid, size_t size, gfp_t gfp_mask);
```

## Arguments

*nid*            the preferred node ID where memory should be allocated

*size*           the number of bytes to allocate

*gfp\_mask*      GFP flags for the allocation

## Description

Like `alloc_pages_exact`, but try to allocate on node `nid` first before falling back.

## Name

`nr_free_zone_pages` — count number of pages beyond high watermark

## Synopsis

```
unsigned long nr_free_zone_pages (int offset);
```

## Arguments

*offset*    The zone index of the highest zone

## Description

`nr_free_zone_pages` counts the number of counts pages which are beyond the high watermark within all zones at or below a given zone index. For each zone, the number of pages is calculated as: `managed_pages - high_pages`

## Name

`nr_free_pagecache_pages` — count number of pages beyond high watermark

## Synopsis

```
unsigned long nr_free_pagecache_pages ( void );
```

## Arguments

*void* no arguments

## Description

`nr_free_pagecache_pages` counts the number of pages which are beyond the high watermark within all zones.

## Name

`find_next_best_node` — find the next node that should appear in a given node's fallback list

## Synopsis

```
int find_next_best_node (int node, nodemask_t * used_node_mask);
```

## Arguments

*node*                      node whose fallback list we're appending

*used\_node\_mask*      nodemask\_t of already used nodes

## Description

We use a number of factors to determine which is the next node that should appear on a given node's fallback list. The node should not have appeared already in *node*'s fallback list, and it should be the next closest node according to the distance array (which contains arbitrary distance values from each node to each node in the system), and should also prefer nodes with no CPUs, since presumably they'll have very little allocation pressure on them otherwise. It returns -1 if no node is found.

## Name

`free_bootmem_with_active_regions` — Call `memblock_free_early_nid` for each active range

## Synopsis

```
void    free_bootmem_with_active_regions    (int    nid,    unsigned    long
max_low_pfn);
```

## Arguments

*nid*                    The node to free memory on. If `MAX_NUMNODES`, all nodes are freed.

*max\_low\_pfn*    The highest PFN that will be passed to `memblock_free_early_nid`

## Description

If an architecture guarantees that all ranges registered contain no holes and may be freed, this this function may be used instead of calling `memblock_free_early_nid` manually.

## Name

`sparse_memory_present_with_active_regions` — Call `memory_present` for each active range

## Synopsis

```
void sparse_memory_present_with_active_regions (int nid);
```

## Arguments

*nid* The node to call `memory_present` for. If `MAX_NUMNODES`, all nodes will be used.

## Description

If an architecture guarantees that all ranges registered contain no holes and may be freed, this function may be used instead of calling `memory_present` manually.



## Name

`get_pfn_range_for_nid` — Return the start and end page frames for a node

## Synopsis

```
void get_pfn_range_for_nid (unsigned int nid, unsigned long * start_pfn,  
unsigned long * end_pfn);
```

## Arguments

<i>nid</i>	The nid to return the range for. If MAX_NUMNODES, the min and max PFN are returned.
<i>start_pfn</i>	Passed by reference. On return, it will have the node start_pfn.
<i>end_pfn</i>	Passed by reference. On return, it will have the node end_pfn.

## Description

It returns the start and end page frame of a node based on information provided by `memblock_set_node`. If called for a node with no available memory, a warning is printed and the start and end PFNs will be 0.

## Name

`absent_pages_in_range` — Return number of page frames in holes within a range

## Synopsis

```
unsigned long absent_pages_in_range (unsigned long start_pfn, unsigned  
long end_pfn);
```

## Arguments

*start\_pfn*    The start PFN to start searching for holes

*end\_pfn*     The end PFN to stop searching for holes

## Description

It returns the number of pages frames in memory holes within a range.

## Name

`node_map_pfn_alignment` — determine the maximum internode alignment

## Synopsis

```
unsigned long node_map_pfn_alignment ( void );
```

## Arguments

*void* no arguments

## Description

This function should be called after node map is populated and sorted. It calculates the maximum power of two alignment which can distinguish all the nodes.

For example, if all nodes are 1GiB and aligned to 1GiB, the return value would indicate 1GiB alignment with  $(1 \ll (30 - \text{PAGE\_SHIFT}))$ . If the nodes are shifted by 256MiB, 256MiB. Note that if only the last node is shifted, 1GiB is enough and this function will indicate so.

This is used to test whether `pfn -> nid` mapping of the chosen memory model has fine enough granularity to avoid incorrect mapping for the populated node map.

Returns the determined alignment in pfn's. 0 if there is no alignment requirement (single node).

## Name

`find_min_pfn_with_active_regions` — Find the minimum PFN registered

## Synopsis

```
unsigned long find_min_pfn_with_active_regions ( void );
```

## Arguments

*void* no arguments

## Description

It returns the minimum PFN based on information provided via `memblock_set_node`.

## Name

`free_area_init_nodes` — Initialise all `pg_data_t` and zone data

## Synopsis

```
void free_area_init_nodes (unsigned long * max_zone_pfn);
```

## Arguments

*max\_zone\_pfn* an array of max PFNs for each zone

## Description

This will call `free_area_init_node` for each active node in the system. Using the page ranges provided by `memblock_set_node`, the size of each zone in each node and their holes is calculated. If the maximum PFN between two adjacent zones match, it is assumed that the zone is empty. For example, if `arch_max_dma_pfn == arch_max_dma32_pfn`, it is assumed that `arch_max_dma32_pfn` has no pages. It is also assumed that a zone starts where the previous one ended. For example, `ZONE_DMA32` starts at `arch_max_dma_pfn`.

## Name

`set_dma_reserve` — set the specified number of pages reserved in the first zone

## Synopsis

```
void set_dma_reserve (unsigned long new_dma_reserve);
```

## Arguments

*new\_dma\_reserve*    The number of pages to mark reserved

## Description

The per-cpu batchsize and zone watermarks are determined by `managed_pages`. In the DMA zone, a significant percentage may be consumed by kernel image and other unfreeable allocations which can skew the watermarks badly. This function may optionally be used to account for unfreeable pages in the first zone (e.g., `ZONE_DMA`). The effect will be lower watermarks and smaller per-cpu batchsize.

## Name

`setup_per_zone_wmarks` — called when `min_free_kbytes` changes or when memory is hot-{added|removed}

## Synopsis

```
void setup_per_zone_wmarks ( void );
```

## Arguments

*void* no arguments

## Description

Ensures that the `watermark[min,low,high]` values for each zone are set correctly with respect to `min_free_kbytes`.

## Name

`get_pfnblock_flags_mask` — Return the requested group of flags for the `pageblock_nr_pages` block of pages

## Synopsis

```
unsigned long get_pfnblock_flags_mask (struct page * page, unsigned long
pfn, unsigned long end_bitidx, unsigned long mask);
```

## Arguments

<i>page</i>	The page within the block of interest
<i>pfn</i>	The target page frame number
<i>end_bitidx</i>	The last bit of interest to retrieve
<i>mask</i>	mask of bits that the caller is interested in

## Return

pageblock\_bits flags



## Name

`set_pfnblock_flags_mask` — Set the requested group of flags for a `pageblock_nr_pages` block of pages

## Synopsis

```
void set_pfnblock_flags_mask (struct page * page, unsigned long flags,  
unsigned long pfn, unsigned long end_bitidx, unsigned long mask);
```

## Arguments

<i>page</i>	The page within the block of interest
<i>flags</i>	The flags to set
<i>pfn</i>	The target page frame number
<i>end_bitidx</i>	The last bit of interest
<i>mask</i>	mask of bits that the caller is interested in

## Name

`alloc_contig_range` — - tries to allocate given range of pages

## Synopsis

```
int alloc_contig_range (unsigned long start, unsigned long end, unsigned
migratetype);
```

## Arguments

<i>start</i>	start PFN to allocate
<i>end</i>	one-past-the-last PFN to allocate
<i>migratetype</i>	migratetype of the underlaying pageblocks (either #MIGRATE_MOVABLE or #MIGRATE_CMA). All pageblocks in range must have the same migratetype and it must be either of the two.

## Description

The PFN range does not have to be pageblock or MAX\_ORDER\_NR\_PAGES aligned, however it's the caller's responsibility to guarantee that we are the only thread that changes migrate type of pageblocks the pages fall in.

The PFN range must belong to a single zone.

Returns zero on success or negative error code. On success all pages which PFN is in [start, end) are allocated for the caller and need to be freed with `free_contig_range`.

## Name

mempool\_destroy — deallocate a memory pool

## Synopsis

```
void mempool_destroy (mempool_t * pool);
```

## Arguments

*pool* pointer to the memory pool which was allocated via mempool\_create.

## Description

Free all reserved elements in *pool* and *pool* itself. This function only sleeps if the `free_fn` function sleeps.

## Name

mempool\_create — create a memory pool

## Synopsis

```
mempool_t * mempool_create (int min_nr, mempool_alloc_t * alloc_fn,  
mempool_free_t * free_fn, void * pool_data);
```

## Arguments

*min\_nr*        the minimum number of elements guaranteed to be allocated for this pool.

*alloc\_fn*     user-defined element-allocation function.

*free\_fn*      user-defined element-freeing function.

*pool\_data*   optional private data available to the user-defined functions.

## Description

this function creates and allocates a guaranteed size, preallocated memory pool. The pool can be used from the `mempool_alloc` and `mempool_free` functions. This function might sleep. Both the `alloc_fn` and the `free_fn` functions might sleep - as long as the `mempool_alloc` function is not called from IRQ contexts.

## Name

`mempool_resize` — resize an existing memory pool

## Synopsis

```
int mempool_resize (mempool_t * pool, int new_min_nr);
```

## Arguments

*pool*                pointer to the memory pool which was allocated via `mempool_create`.

*new\_min\_nr*        the new minimum number of elements guaranteed to be allocated for this pool.

## Description

This function shrinks/grows the pool. In the case of growing, it cannot be guaranteed that the pool will be grown to the new size immediately, but new `mempool_free` calls will refill it. This function may sleep.

Note, the caller must guarantee that no `mempool_destroy` is called while this function is running. `mempool_alloc` & `mempool_free` might be called (eg. from IRQ contexts) while this function executes.

## Name

`mempool_alloc` — allocate an element from a specific memory pool

## Synopsis

```
void * mempool_alloc (mempool_t * pool, gfp_t gfp_mask);
```

## Arguments

*pool*            pointer to the memory pool which was allocated via `mempool_create`.

*gfp\_mask*    the usual allocation bitmask.

## Description

this function only sleeps if the `alloc_fn` function sleeps or returns NULL. Note that due to preallocation, this function *\*never\** fails when called from process contexts. (it might fail if called from an IRQ context.)

## Note

using `__GFP_ZERO` is not supported.

## Name

`mempool_free` — return an element to the pool.

## Synopsis

```
void mempool_free (void * element, mempool_t * pool);
```

## Arguments

*element*    pool element pointer.

*pool*        pointer to the memory pool which was allocated via `mempool_create`.

## Description

this function only sleeps if the `free_fn` function sleeps.

## Name

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

## Synopsis

```
struct dma_pool * dma_pool_create (const char * name, struct device *  
dev, size_t size, size_t align, size_t boundary);
```

## Arguments

<i>name</i>	name of pool, for diagnostics
<i>dev</i>	device that will be doing the DMA
<i>size</i>	size of the blocks in this pool.
<i>align</i>	alignment requirement for blocks; must be a power of two
<i>boundary</i>	returned blocks won't cross this power of two boundary

## Context

`!in_interrupt`

## Description

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

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



## Name

`dma_pool_destroy` — destroys a pool of dma memory blocks.

## Synopsis

```
void dma_pool_destroy (struct dma_pool * pool);
```

## Arguments

*pool*    dma pool that will be destroyed

## Context

`!in_interrupt`

## Description

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

## Name

`dma_pool_alloc` — get a block of consistent memory

## Synopsis

```
void * dma_pool_alloc (struct dma_pool * pool, gfp_t mem_flags,  
dma_addr_t * handle);
```

## Arguments

*pool*            dma pool that will produce the block

*mem\_flags*    GFP\_\* bitmask

*handle*        pointer to dma address of block

## Description

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

## Name

`dma_pool_free` — put block back into dma pool

## Synopsis

```
void dma_pool_free (struct dma_pool * pool, void * vaddr, dma_addr_t  
dma);
```

## Arguments

*pool*     the dma pool holding the block

*vaddr*    virtual address of block

*dma*      dma address of block

## Description

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

## Name

`dmam_pool_create` — Managed `dma_pool_create`

## Synopsis

```
struct dma_pool * dmam_pool_create (const char * name, struct device *  
dev, size_t size, size_t align, size_t allocation);
```

## Arguments

<i>name</i>	name of pool, for diagnostics
<i>dev</i>	device that will be doing the DMA
<i>size</i>	size of the blocks in this pool.
<i>align</i>	alignment requirement for blocks; must be a power of two
<i>allocation</i>	returned blocks won't cross this boundary (or zero)

## Description

Managed `dma_pool_create`. DMA pool created with this function is automatically destroyed on driver detach.

## Name

`dmam_pool_destroy` — Managed `dma_pool_destroy`

## Synopsis

```
void dmam_pool_destroy (struct dma_pool * pool);
```

## Arguments

*pool* dma pool that will be destroyed

## Description

Managed `dma_pool_destroy`.

## Name

`balance_dirty_pages_ratelimited` — balance dirty memory state

## Synopsis

```
void balance_dirty_pages_ratelimited (struct address_space * mapping);
```

## Arguments

*mapping*    `address_space` which was dirtied

## Description

Processes which are dirtying memory should call in here once for each page which was newly dirtied. The function will periodically check the system's dirty state and will initiate writeback if needed.

On really big machines, `get_writeback_state` is expensive, so try to avoid calling it too often (ratelimiting). But once we're over the dirty memory limit we decrease the ratelimiting by a lot, to prevent individual processes from overshooting the limit by (`ratelimit_pages`) each.

## Name

`tag_pages_for_writeback` — tag pages to be written by `write_cache_pages`

## Synopsis

```
void tag_pages_for_writeback (struct address_space * mapping, pgoff_t
start, pgoff_t end);
```

## Arguments

*mapping*    address space structure to write

*start*      starting page index

*end*        ending page index (inclusive)

## Description

This function scans the page range from *start* to *end* (inclusive) and tags all pages that have DIRTY tag set with a special TOWRITE tag. The idea is that `write_cache_pages` (or whoever calls this function) will then use TOWRITE tag to identify pages eligible for writeback. This mechanism is used to avoid livelocking of writeback by a process steadily creating new dirty pages in the file (thus it is important for this function to be quick so that it can tag pages faster than a dirtying process can create them).

## Name

`write_cache_pages` — walk the list of dirty pages of the given address space and write all of them.

## Synopsis

```
int write_cache_pages (struct address_space * mapping, struct
writeback_control * wbc, writepage_t writepage, void * data);
```

## Arguments

<i>mapping</i>	address space structure to write
<i>wbc</i>	subtract the number of written pages from <i>*wbc-&gt;nr_to_write</i>
<i>writepage</i>	function called for each page
<i>data</i>	data passed to writepage function

## Description

If a page is already under I/O, `write_cache_pages` skips it, even if it's dirty. This is desirable behaviour for memory-cleaning writeback, but it is INCORRECT for data-integrity system calls such as `fsync`. `fsync` and `msync` need to guarantee that all the data which was dirty at the time the call was made get new I/O started against them. If `wbc->sync_mode` is `WB_SYNC_ALL` then we were called for data integrity and we must wait for existing IO to complete.

To avoid livelocks (when other process dirties new pages), we first tag pages which should be written back with `TOWRITE` tag and only then start writing them. For data-integrity sync we have to be careful so that we do not miss some pages (e.g., because some other process has cleared `TOWRITE` tag we set). The rule we follow is that `TOWRITE` tag can be cleared only by the process clearing the `DIRTY` tag (and submitting the page for IO).



## Name

`generic_writepages` — walk the list of dirty pages of the given address space and `writepage` all of them.

## Synopsis

```
int generic_writepages (struct address_space * mapping, struct
writeback_control * wbc);
```

## Arguments

*mapping* address space structure to write

*wbc* subtract the number of written pages from `*wbc->nr_to_write`

## Description

This is a library function, which implements the `writepages` `address_space_operation`.

## Name

`write_one_page` — write out a single page and optionally wait on I/O

## Synopsis

```
int write_one_page (struct page * page, int wait);
```

## Arguments

*page* the page to write

*wait* if true, wait on writeout

## Description

The page must be locked by the caller and will be unlocked upon return.

`write_one_page` returns a negative error code if I/O failed.

## Name

`wait_for_stable_page` — wait for writeback to finish, if necessary.

## Synopsis

```
void wait_for_stable_page (struct page * page);
```

## Arguments

*page*    The page to wait on.

## Description

This function determines if the given page is related to a backing device that requires page contents to be held stable during writeback. If so, then it will wait for any pending writeback to complete.

## Name

`truncate_inode_pages_range` — truncate range of pages specified by start & end byte offsets

## Synopsis

```
void truncate_inode_pages_range (struct address_space * mapping, loff_t
    lstart, loff_t lend);
```

## Arguments

*mapping*    mapping to truncate

*lstart*     offset from which to truncate

*lend*       offset to which to truncate (inclusive)

## Description

Truncate the page cache, removing the pages that are between specified offsets (and zeroing out partial pages if *lstart* or *lend* + 1 is not page aligned).

Truncate takes two passes - the first pass is nonblocking. It will not block on page locks and it will not block on writeback. The second pass will wait. This is to prevent as much IO as possible in the affected region. The first pass will remove most pages, so the search cost of the second pass is low.

We pass down the cache-hot hint to the page freeing code. Even if the mapping is large, it is probably the case that the final pages are the most recently touched, and freeing happens in ascending file offset order.

Note that since `->invalidatepage` accepts range to invalidate `truncate_inode_pages_range` is able to handle cases where *lend* + 1 is not page aligned properly.

## Name

`truncate_inode_pages` — truncate \*all\* the pages from an offset

## Synopsis

```
void truncate_inode_pages (struct address_space * mapping, loff_t  
lstart);
```

## Arguments

*mapping* mapping to truncate

*lstart* offset from which to truncate

## Description

Called under (and serialised by) `inode->i_mutex`.

## Note

When this function returns, there can be a page in the process of deletion (inside `__delete_from_page_cache`) in the specified range. Thus `mapping->npages` can be non-zero when this function returns even after truncation of the whole mapping.

## Name

`truncate_inode_pages_final` — truncate *\*all\** pages before inode dies

## Synopsis

```
void truncate_inode_pages_final (struct address_space * mapping);
```

## Arguments

*mapping* mapping to truncate

## Description

Called under (and serialized by) `inode->i_mutex`.

Filesystems have to use this in the `.evict_inode` path to inform the VM that this is the final truncate and the inode is going away.

## Name

`invalidate_mapping_pages` — Invalidate all the unlocked pages of one inode

## Synopsis

```
unsigned long invalidate_mapping_pages (struct address_space * mapping,  
pgoff_t start, pgoff_t end);
```

## Arguments

*mapping*    the `address_space` which holds the pages to invalidate

*start*      the offset 'from' which to invalidate

*end*        the offset 'to' which to invalidate (inclusive)

## Description

This function only removes the unlocked pages, if you want to remove all the pages of one inode, you must call `truncate_inode_pages`.

`invalidate_mapping_pages` will not block on IO activity. It will not invalidate pages which are dirty, locked, under writeback or mapped into pagetables.

## Name

`invalidate_inode_pages2_range` — remove range of pages from an `address_space`

## Synopsis

```
int  invalidate_inode_pages2_range (struct address_space * mapping,
pgoff_t start, pgoff_t end);
```

## Arguments

*mapping*    the `address_space`

*start*       the page offset 'from' which to invalidate

*end*         the page offset 'to' which to invalidate (inclusive)

## Description

Any pages which are found to be mapped into pagetables are unmapped prior to invalidation.

Returns `-EBUSY` if any pages could not be invalidated.



## Name

`invalidate_inode_pages2` — remove all pages from an `address_space`

## Synopsis

```
int invalidate_inode_pages2 (struct address_space * mapping);
```

## Arguments

*mapping* the `address_space`

## Description

Any pages which are found to be mapped into pagetables are unmapped prior to invalidation.

Returns `-EBUSY` if any pages could not be invalidated.

## Name

`truncate_pagecache` — unmap and remove pagecache that has been truncated

## Synopsis

```
void truncate_pagecache (struct inode * inode, loff_t newsize);
```

## Arguments

*inode*      inode

*newsize*    new file size

## Description

inode's new `i_size` must already be written before `truncate_pagecache` is called.

This function should typically be called before the filesystem releases resources associated with the freed range (eg. deallocates blocks). This way, pagecache will always stay logically coherent with on-disk format, and the filesystem would not have to deal with situations such as `writepage` being called for a page that has already had its underlying blocks deallocated.

## Name

`truncate_setsize` — update inode and pagecache for a new file size

## Synopsis

```
void truncate_setsize (struct inode * inode, loff_t newsize);
```

## Arguments

*inode*      inode

*newsize*    new file size

## Description

`truncate_setsize` updates `i_size` and performs pagecache truncation (if necessary) to *newsize*. It will be typically be called from the filesystem's `setattr` function when `ATTR_SIZE` is passed in.

Must be called with a lock serializing truncates and writes (generally `i_mutex` but e.g. xfs uses a different lock) and before all filesystem specific block truncation has been performed.

## Name

pagecache\_isize\_extended — update pagecache after extension of `i_size`

## Synopsis

```
void pagecache_isize_extended (struct inode * inode, loff_t from, loff_t to);
```

## Arguments

*inode*    inode for which `i_size` was extended

*from*    original inode size

*to*       new inode size

## Description

Handle extension of inode size either caused by extending truncate or by write starting after current `i_size`. We mark the page straddling current `i_size` RO so that `page_mkwrite` is called on the nearest write access to the page. This way filesystem can be sure that `page_mkwrite` is called on the page before user writes to the page via `mmap` after the `i_size` has been changed.

The function must be called after `i_size` is updated so that page fault coming after we unlock the page will already see the new `i_size`. The function must be called while we still hold `i_mutex` - this not only makes sure `i_size` is stable but also that userspace cannot observe new `i_size` value before we are prepared to store `mmap` writes at new inode size.

## Name

`truncate_pagecache_range` — unmap and remove pagecache that is hole-punched

## Synopsis

```
void truncate_pagecache_range (struct inode * inode, loff_t lstart,  
                                loff_t lend);
```

## Arguments

*inode*     inode

*lstart*    offset of beginning of hole

*lend*      offset of last byte of hole

## Description

This function should typically be called before the filesystem releases resources associated with the freed range (eg. deallocates blocks). This way, pagecache will always stay logically coherent with on-disk format, and the filesystem would not have to deal with situations such as writepage being called for a page that has already had its underlying blocks deallocated.

---

# **Chapter 5. Kernel IPC facilities**

## **IPC utilities**

## Name

`ipc_init` — initialise ipc subsystem

## Synopsis

```
int ipc_init ( void );
```

## Arguments

*void* no arguments

## Description

The various sysv ipc resources (semaphores, messages and shared memory) are initialised.

A callback routine is registered into the memory hotplug notifier

## chain

since msgmni scales to lowmem this callback routine will be called upon successful memory add / remove to recompute msgmni.

## Name

`ipc_init_ids` — initialise ipc identifiers

## Synopsis

```
void ipc_init_ids (struct ipc_ids * ids);
```

## Arguments

*ids* ipc identifier set

## Description

Set up the sequence range to use for the ipc identifier range (limited below IPCMNI) then initialise the `ids` idr.



## Name

`ipc_init_proc_interface` — create a proc interface for sysipc types using a `seq_file` interface.

## Synopsis

```
void ipc_init_proc_interface (const char * path, const char * header,  
int ids, int (*show) (struct seq_file *, void *));
```

## Arguments

<i>path</i>	Path in procfs
<i>header</i>	Banner to be printed at the beginning of the file.
<i>ids</i>	ipc id table to iterate.
<i>show</i>	show routine.

## Name

`ipc_findkey` — find a key in an ipc identifier set

## Synopsis

```
struct kern_ipc_perm * ipc_findkey (struct ipc_ids * ids, key_t key);
```

## Arguments

*ids* ipc identifier set

*key* key to find

## Description

Returns the locked pointer to the ipc structure if found or NULL otherwise. If key is found ipc points to the owning ipc structure

Called with `ipc_ids.rwsem` held.

## Name

`ipc_get_maxid` — get the last assigned id

## Synopsis

```
int ipc_get_maxid (struct ipc_ids * ids);
```

## Arguments

*ids* ipc identifier set

## Description

Called with `ipc_ids.rwsem` held.

## Name

`ipc_addid` — add an ipc identifier

## Synopsis

```
int ipc_addid (struct ipc_ids * ids, struct kern_ipc_perm * new, int
size);
```

## Arguments

*ids*    ipc identifier set

*new*    new ipc permission set

*size*    limit for the number of used ids

## Description

Add an entry 'new' to the ipc ids idr. The permissions object is initialised and the first free entry is set up and the id assigned is returned. The 'new' entry is returned in a locked state on success. On failure the entry is not locked and a negative err-code is returned.

Called with writer `ipc_ids.rwsem` held.

## Name

`ipcget_new` — create a new ipc object

## Synopsis

```
int ipcget_new (struct ipc_namespace * ns, struct ipc_ids * ids, const
struct ipc_ops * ops, struct ipc_params * params);
```

## Arguments

<i>ns</i>	ipc namespace
<i>ids</i>	ipc identifier set
<i>ops</i>	the actual creation routine to call
<i>params</i>	its parameters

## Description

This routine is called by `sys_msgget`, `sys_semget` and `sys_shmget` when the key is `IPC_PRIVATE`.

## Name

`ipc_check_perms` — check security and permissions for an ipc object

## Synopsis

```
int ipc_check_perms (struct ipc_namespace * ns, struct kern_ipc_perm *  
ipcp, const struct ipc_ops * ops, struct ipc_params * params);
```

## Arguments

<i>ns</i>	ipc namespace
<i>ipcp</i>	ipc permission set
<i>ops</i>	the actual security routine to call
<i>params</i>	its parameters

## Description

This routine is called by `sys_msgget`, `sys_semget` and `sys_shmget` when the key is not `IPC_PRIVATE` and that key already exists in the ds IDR.

On success, the ipc id is returned.

It is called with `ipc_ids.rwsem` and `ipcp->lock` held.

## Name

`ipcget_public` — get an ipc object or create a new one

## Synopsis

```
int ipcget_public (struct ipc_namespace * ns, struct ipc_ids * ids,  
const struct ipc_ops * ops, struct ipc_params * params);
```

## Arguments

<i>ns</i>	ipc namespace
<i>ids</i>	ipc identifier set
<i>ops</i>	the actual creation routine to call
<i>params</i>	its parameters

## Description

This routine is called by `sys_msgget`, `sys_semget` and `sys_shmget` when the key is not `IPC_PRIVATE`. It adds a new entry if the key is not found and does some permission / security checkings if the key is found.

On success, the ipc id is returned.

## Name

`ipc_rmid` — remove an ipc identifier

## Synopsis

```
void ipc_rmid (struct ipc_ids * ids, struct kern_ipc_perm * ipcp);
```

## Arguments

*ids*    ipc identifier set

*ipcp*   ipc perm structure containing the identifier to remove

## Description

`ipc_ids.rwsem` (as a writer) and the spinlock for this ID are held before this function is called, and remain locked on the exit.



## Name

`ipc_alloc` — allocate ipc space

## Synopsis

```
void * ipc_alloc (int size);
```

## Arguments

*size*    size desired

## Description

Allocate memory from the appropriate pools and return a pointer to it. NULL is returned if the allocation fails

## Name

`ipc_free` — free ipc space

## Synopsis

```
void ipc_free (void * ptr, int size);
```

## Arguments

*ptr*     pointer returned by `ipc_alloc`

*size*    size of block

## Description

Free a block created with `ipc_alloc`. The caller must know the size used in the allocation call.

## Name

`ipc_rcu_alloc` — allocate ipc and rcu space

## Synopsis

```
void * ipc_rcu_alloc (int size);
```

## Arguments

*size*    size desired

## Description

Allocate memory for the rcu header structure + the object. Returns the pointer to the object or NULL upon failure.

## Name

`ipcperms` — check ipc permissions

## Synopsis

```
int ipcperms (struct ipc_namespace * ns, struct kern_ipc_perm * ipcp,
short flag);
```

## Arguments

*ns* ipc namespace

*ipcp* ipc permission set

*flag* desired permission set

## Description

Check user, group, other permissions for access to ipc resources. return 0 if allowed

*flag* will most probably be 0 or S\_...UGO from <linux/stat.h>

## Name

`kernel_to_ipc64_perm` — convert kernel ipc permissions to user

## Synopsis

```
void kernel_to_ipc64_perm (struct kern_ipc_perm * in, struct ipc64_perm  
* out);
```

## Arguments

*in*    kernel permissions

*out*   new style ipc permissions

## Description

Turn the kernel object *in* into a set of permissions descriptions for returning to userspace (*out*).

## Name

`ipc64_perm_to_ipc_perm` — convert new ipc permissions to old

## Synopsis

```
void ipc64_perm_to_ipc_perm (struct ipc64_perm * in, struct ipc_perm  
* out);
```

## Arguments

*in*    new style ipc permissions

*out*   old style ipc permissions

## Description

Turn the new style permissions object *in* into a compatibility object and store it into the *out* pointer.

## Name

`ipc_obtain_object_idr` —

## Synopsis

```
struct kern_ipc_perm * ipc_obtain_object_idr (struct ipc_ids * ids, int
id);
```

## Arguments

*ids* ipc identifier set

*id* ipc id to look for

## Description

Look for an id in the ipc ids idr and return associated ipc object.

Call inside the RCU critical section. The ipc object is *\*not\** locked on exit.

## Name

`ipc_lock` — lock an ipc structure without rwsem held

## Synopsis

```
struct kern_ipc_perm * ipc_lock (struct ipc_ids * ids, int id);
```

## Arguments

*ids* ipc identifier set

*id* ipc id to look for

## Description

Look for an id in the ipc ids idr and lock the associated ipc object.

The ipc object is locked on successful exit.



## Name

`ipc_obtain_object_check` —

## Synopsis

```
struct kern_ipc_perm * ipc_obtain_object_check (struct ipc_ids * ids,  
int id);
```

## Arguments

*ids* ipc identifier set

*id* ipc id to look for

## Description

Similar to `ipc_obtain_object_idr` but also checks the ipc object reference counter.

Call inside the RCU critical section. The ipc object is *\*not\** locked on exit.

## Name

ipcget — Common sys\_\*get code

## Synopsis

```
int ipcget (struct ipc_namespace * ns, struct ipc_ids * ids, const
struct ipc_ops * ops, struct ipc_params * params);
```

## Arguments

*ns* namespace

*ids* ipc identifier set

*ops* operations to be called on ipc object creation, permission checks and further checks

*params* the parameters needed by the previous operations.

## Description

Common routine called by `sys_msgget`, `sys_semget` and `sys_shmget`.

## Name

`ipc_update_perm` — update the permissions of an ipc object

## Synopsis

```
int ipc_update_perm (struct ipc64_perm * in, struct kern_ipc_perm *  
out);
```

## Arguments

*in*    the permission given as input.

*out*   the permission of the ipc to set.

## Name

`ipcctl_pre_down_nolock` — retrieve an ipc and check permissions for some `IPC_XXX` cmd

## Synopsis

```
struct kern_ipc_perm * ipcctl_pre_down_nolock (struct ipc_namespace *  
ns, struct ipc_ids * ids, int id, int cmd, struct ipc64_perm * perm,  
int extra_perm);
```

## Arguments

<i>ns</i>	ipc namespace
<i>ids</i>	the table of ids where to look for the ipc
<i>id</i>	the id of the ipc to retrieve
<i>cmd</i>	the cmd to check
<i>perm</i>	the permission to set
<i>extra_perm</i>	one extra permission parameter used by msq

## Description

This function does some common audit and permissions check for some `IPC_XXX` cmd and is called from `semctl_down`, `shmctl_down` and `msgctl_down`. It must be called without any lock held and - retrieves the ipc with the given id in the given table. - performs some audit and permission check, depending on the given cmd - returns a pointer to the ipc object or otherwise, the corresponding error.

Call holding the both the rwsem and the rcu read lock.

## Name

`ipc_parse_version` — ipc call version

## Synopsis

```
int ipc_parse_version (int * cmd);
```

## Arguments

*cmd* pointer to command

## Description

Return `IPC_64` for new style IPC and `IPC_OLD` for old style IPC. The *cmd* value is turned from an encoding command and version into just the command code.

---

# Chapter 6. FIFO Buffer

## kfifo interface

## Name

DECLARE\_KFIFO\_PTR — macro to declare a fifo pointer object

## Synopsis

```
DECLARE_KFIFO_PTR ( fifo, type );
```

## Arguments

*fifo*    name of the declared fifo

*type*    type of the fifo elements

## Name

DECLARE\_KFIFO — macro to declare a fifo object

## Synopsis

```
DECLARE_KFIFO ( fifo, type, size);
```

## Arguments

*fifo*    name of the declared fifo

*type*    type of the fifo elements

*size*    the number of elements in the fifo, this must be a power of 2



## Name

INIT\_KFIFO — Initialize a fifo declared by DECLARE\_KFIFO

## Synopsis

```
INIT_KFIFO ( fifo );
```

## Arguments

*fifo*   name of the declared fifo datatype

## Name

DEFINE\_KFIFO — macro to define and initialize a fifo

## Synopsis

```
DEFINE_KFIFO ( fifo, type, size );
```

## Arguments

*fifo* name of the declared fifo datatype

*type* type of the fifo elements

*size* the number of elements in the fifo, this must be a power of 2

## Note

the macro can be used for global and local fifo data type variables.

## Name

`kfifo_initialized` — Check if the fifo is initialized

## Synopsis

```
kfifo_initialized ( fifo );
```

## Arguments

*fifo* address of the fifo to check

## Description

Return `true` if fifo is initialized, otherwise `false`. Assumes the fifo was 0 before.

## Name

`kfifo_esize` — returns the size of the element managed by the fifo

## Synopsis

```
kfifo_esize ( fifo );
```

## Arguments

*fifo* address of the fifo to be used

## Name

`kfifo_recsz` — returns the size of the record length field

## Synopsis

```
kfifo_recsz ( fifo );
```

## Arguments

*fifo* address of the fifo to be used

## Name

`kfifo_size` — returns the size of the fifo in elements

## Synopsis

```
kfifo_size ( fifo );
```

## Arguments

*fifo*    address of the fifo to be used

## Name

`kfifo_reset` — removes the entire fifo content

## Synopsis

```
kfifo_reset ( fifo );
```

## Arguments

*fifo* address of the fifo to be used

## Note

usage of `kfifo_reset` is dangerous. It should be only called when the fifo is exclusived locked or when it is secured that no other thread is accessing the fifo.

## Name

`kfifo_reset_out` — skip fifo content

## Synopsis

```
kfifo_reset_out ( fifo );
```

## Arguments

*fifo* address of the fifo to be used

## Note

The usage of `kfifo_reset_out` is safe until it will be only called from the reader thread and there is only one concurrent reader. Otherwise it is dangerous and must be handled in the same way as `kfifo_reset`.



## Name

`kfifo_len` — returns the number of used elements in the fifo

## Synopsis

```
kfifo_len ( fifo );
```

## Arguments

*fifo* address of the fifo to be used

## Name

`kfifo_is_empty` — returns true if the fifo is empty

## Synopsis

```
kfifo_is_empty ( fifo );
```

## Arguments

*fifo*    address of the fifo to be used

## Name

`kfifo_is_full` — returns true if the fifo is full

## Synopsis

```
kfifo_is_full ( fifo );
```

## Arguments

*fifo*    address of the fifo to be used

## Name

`kfifo_avail` — returns the number of unused elements in the fifo

## Synopsis

```
kfifo_avail ( fifo );
```

## Arguments

*fifo* address of the fifo to be used

## Name

`kfifo_skip` — skip output data

## Synopsis

```
kfifo_skip ( fifo );
```

## Arguments

*fifo* address of the fifo to be used

## Name

`kfifo_peek_len` — gets the size of the next fifo record

## Synopsis

```
kfifo_peek_len ( fifo );
```

## Arguments

*fifo* address of the fifo to be used

## Description

This function returns the size of the next fifo record in number of bytes.

## Name

`kfifo_alloc` — dynamically allocates a new fifo buffer

## Synopsis

```
kfifo_alloc ( fifo, size, gfp_mask );
```

## Arguments

*fifo*            pointer to the fifo

*size*            the number of elements in the fifo, this must be a power of 2

*gfp\_mask*    `get_free_pages` mask, passed to `kmalloc`

## Description

This macro dynamically allocates a new fifo buffer.

The number of elements will be rounded-up to a power of 2. The fifo will be released with `kfifo_free`. Return 0 if no error, otherwise an error code.

## Name

`kfifo_free` — frees the fifo

## Synopsis

```
kfifo_free ( fifo );
```

## Arguments

*fifo* the fifo to be freed



## Name

`kfifo_init` — initialize a fifo using a preallocated buffer

## Synopsis

```
kfifo_init ( fifo, buffer, size );
```

## Arguments

*fifo*      the fifo to assign the buffer

*buffer*   the preallocated buffer to be used

*size*      the size of the internal buffer, this have to be a power of 2

## Description

This macro initialize a fifo using a preallocated buffer.

The numer of elements will be rounded-up to a power of 2. Return 0 if no error, otherwise an error code.

## Name

`kfifo_put` — put data into the fifo

## Synopsis

```
kfifo_put ( fifo, val );
```

## Arguments

*fifo*    address of the fifo to be used

*val*    the data to be added

## Description

This macro copies the given value into the fifo. It returns 0 if the fifo was full. Otherwise it returns the number processed elements.

Note that with only one concurrent reader and one concurrent writer, you don't need extra locking to use these macro.

## Name

`kfifo_get` — get data from the fifo

## Synopsis

```
kfifo_get ( fifo, val);
```

## Arguments

*fifo*    address of the fifo to be used

*val*    address where to store the data

## Description

This macro reads the data from the fifo. It returns 0 if the fifo was empty. Otherwise it returns the number processed elements.

Note that with only one concurrent reader and one concurrent writer, you don't need extra locking to use these macro.

## Name

`kfifo_peek` — get data from the fifo without removing

## Synopsis

```
kfifo_peek ( fifo, val);
```

## Arguments

*fifo*    address of the fifo to be used

*val*    address where to store the data

## Description

This reads the data from the fifo without removing it from the fifo. It returns 0 if the fifo was empty. Otherwise it returns the number processed elements.

Note that with only one concurrent reader and one concurrent writer, you don't need extra locking to use these macro.

## Name

`kfifo_in` — put data into the fifo

## Synopsis

```
kfifo_in ( fifo, buf, n );
```

## Arguments

*fifo*    address of the fifo to be used

*buf*    the data to be added

*n*      number of elements to be added

## Description

This macro copies the given buffer into the fifo and returns the number of copied elements.

Note that with only one concurrent reader and one concurrent writer, you don't need extra locking to use these macro.

## Name

`kfifo_in_spinlocked` — put data into the fifo using a spinlock for locking

## Synopsis

```
kfifo_in_spinlocked ( fifo, buf, n, lock );
```

## Arguments

*fifo*    address of the fifo to be used

*buf*    the data to be added

*n*      number of elements to be added

*lock*   pointer to the spinlock to use for locking

## Description

This macro copies the given values buffer into the fifo and returns the number of copied elements.

## Name

`kfifo_out` — get data from the fifo

## Synopsis

```
kfifo_out ( fifo, buf, n );
```

## Arguments

*fifo*    address of the fifo to be used

*buf*    pointer to the storage buffer

*n*       max. number of elements to get

## Description

This macro get some data from the fifo and return the numbers of elements copied.

Note that with only one concurrent reader and one concurrent writer, you don't need extra locking to use these macro.

## Name

`kfifo_out_spinlocked` — get data from the fifo using a spinlock for locking

## Synopsis

```
kfifo_out_spinlocked ( fifo, buf, n, lock );
```

## Arguments

*fifo*    address of the fifo to be used

*buf*    pointer to the storage buffer

*n*       max. number of elements to get

*lock*   pointer to the spinlock to use for locking

## Description

This macro get the data from the fifo and return the numbers of elements copied.



## Name

`kfifo_from_user` — puts some data from user space into the fifo

## Synopsis

```
kfifo_from_user ( fifo, from, len, copied );
```

## Arguments

*fifo*      address of the fifo to be used

*from*     pointer to the data to be added

*len*       the length of the data to be added

*copied*   pointer to output variable to store the number of copied bytes

## Description

This macro copies at most *len* bytes from the *from* into the fifo, depending of the available space and returns `-EFAULT/0`.

Note that with only one concurrent reader and one concurrent writer, you don't need extra locking to use these macro.

## Name

`kfifo_to_user` — copies data from the fifo into user space

## Synopsis

```
kfifo_to_user ( fifo, to, len, copied );
```

## Arguments

*fifo*      address of the fifo to be used

*to*        where the data must be copied

*len*       the size of the destination buffer

*copied*   pointer to output variable to store the number of copied bytes

## Description

This macro copies at most *len* bytes from the fifo into the *to* buffer and returns `-EFAULT/0`.

Note that with only one concurrent reader and one concurrent writer, you don't need extra locking to use these macro.

## Name

`kfifo_dma_in_prepare` — setup a scatterlist for DMA input

## Synopsis

```
kfifo_dma_in_prepare ( fifo, sgl, nents, len );
```

## Arguments

<i>fifo</i>	address of the fifo to be used
<i>sgl</i>	pointer to the scatterlist array
<i>nents</i>	number of entries in the scatterlist array
<i>len</i>	number of elements to transfer

## Description

This macro fills a scatterlist for DMA input. It returns the number entries in the scatterlist array.

Note that with only one concurrent reader and one concurrent writer, you don't need extra locking to use these macros.

## Name

`kfifo_dma_in_finish` — finish a DMA IN operation

## Synopsis

```
kfifo_dma_in_finish ( fifo, len );
```

## Arguments

*fifo*    address of the fifo to be used

*len*     number of bytes to received

## Description

This macro finish a DMA IN operation. The in counter will be updated by the len parameter. No error checking will be done.

Note that with only one concurrent reader and one concurrent writer, you don't need extra locking to use these macros.

## Name

`kfifo_dma_out_prepare` — setup a scatterlist for DMA output

## Synopsis

```
kfifo_dma_out_prepare ( fifo, sgl, nents, len );
```

## Arguments

<i>fifo</i>	address of the fifo to be used
<i>sgl</i>	pointer to the scatterlist array
<i>nents</i>	number of entries in the scatterlist array
<i>len</i>	number of elements to transfer

## Description

This macro fills a scatterlist for DMA output which at most *len* bytes to transfer. It returns the number entries in the scatterlist array. A zero means there is no space available and the scatterlist is not filled.

Note that with only one concurrent reader and one concurrent writer, you don't need extra locking to use these macros.

## Name

`kfifo_dma_out_finish` — finish a DMA OUT operation

## Synopsis

```
kfifo_dma_out_finish ( fifo, len);
```

## Arguments

*fifo*    address of the fifo to be used

*len*     number of bytes transferred

## Description

This macro finish a DMA OUT operation. The out counter will be updated by the len parameter. No error checking will be done.

Note that with only one concurrent reader and one concurrent writer, you don't need extra locking to use these macros.

## Name

`kfifo_out_peek` — gets some data from the fifo

## Synopsis

```
kfifo_out_peek ( fifo, buf, n );
```

## Arguments

*fifo*    address of the fifo to be used

*buf*    pointer to the storage buffer

*n*       max. number of elements to get

## Description

This macro get the data from the fifo and return the numbers of elements copied. The data is not removed from the fifo.

Note that with only one concurrent reader and one concurrent writer, you don't need extra locking to use these macro.

---

# Chapter 7. relay interface support

Relay interface support is designed to provide an efficient mechanism for tools and facilities to relay large amounts of data from kernel space to user space.

## relay interface



## Name

`relay_buf_full` — boolean, is the channel buffer full?

## Synopsis

```
int relay_buf_full (struct rchan_buf * buf);
```

## Arguments

*buf*    channel buffer

## Description

Returns 1 if the buffer is full, 0 otherwise.

## Name

`relay_reset` — reset the channel

## Synopsis

```
void relay_reset (struct rchan * chan);
```

## Arguments

*chan* the channel

## Description

This has the effect of erasing all data from all channel buffers and restarting the channel in its initial state. The buffers are not freed, so any mappings are still in effect.

NOTE. Care should be taken that the channel isn't actually being used by anything when this call is made.

## Name

`relay_open` — create a new relay channel

## Synopsis

```
struct rchan * relay_open (const char * base_filename, struct dentry
* parent, size_t subbuf_size, size_t n_subbufs, struct rchan_callbacks
* cb, void * private_data);
```

## Arguments

<i>base_filename</i>	base name of files to create, NULL for buffering only
<i>parent</i>	dentry of parent directory, NULL for root directory or buffer
<i>subbuf_size</i>	size of sub-buffers
<i>n_subbufs</i>	number of sub-buffers
<i>cb</i>	client callback functions
<i>private_data</i>	user-defined data

## Description

Returns channel pointer if successful, NULL otherwise.

Creates a channel buffer for each cpu using the sizes and attributes specified. The created channel buffer files will be named `base_filename0...base_filenameN-1`. File permissions will be `S_IRUSR`.

## Name

`relay_switch_subbuf` — switch to a new sub-buffer

## Synopsis

```
size_t relay_switch_subbuf (struct rchan_buf * buf, size_t length);
```

## Arguments

*buf*        channel buffer

*length*    size of current event

## Description

Returns either the length passed in or 0 if full.

Performs sub-buffer-switch tasks such as invoking callbacks, updating padding counts, waking up readers, etc.

## Name

`relay_subbufs_consumed` — update the buffer's sub-buffers-consumed count

## Synopsis

```
void relay_subbufs_consumed (struct rchan * chan, unsigned int cpu,
size_t subbufs_consumed);
```

## Arguments

<i>chan</i>	the channel
<i>cpu</i>	the cpu associated with the channel buffer to update
<i>subbufs_consumed</i>	number of sub-buffers to add to current buf's count

## Description

Adds to the channel buffer's consumed sub-buffer count. `subbufs_consumed` should be the number of sub-buffers newly consumed, not the total consumed.

NOTE. Kernel clients don't need to call this function if the channel mode is 'overwrite'.

## Name

`relay_close` — close the channel

## Synopsis

```
void relay_close (struct rchan * chan);
```

## Arguments

*chan* the channel

## Description

Closes all channel buffers and frees the channel.

## Name

relay\_flush — close the channel

## Synopsis

```
void relay_flush (struct rchan * chan);
```

## Arguments

*chan* the channel

## Description

Flushes all channel buffers, i.e. forces buffer switch.

## Name

`relay_mmap_buf` — mmap channel buffer to process address space

## Synopsis

```
int relay_mmap_buf (struct rchan_buf * buf, struct vm_area_struct *  
vma);
```

## Arguments

*buf* relay channel buffer

*vma* vm\_area\_struct describing memory to be mapped

## Description

Returns 0 if ok, negative on error

Caller should already have grabbed `mmap_sem`.



## Name

`relay_alloc_buf` — allocate a channel buffer

## Synopsis

```
void * relay_alloc_buf (struct rchan_buf * buf, size_t * size);
```

## Arguments

*buf*     the buffer struct

*size*    total size of the buffer

## Description

Returns a pointer to the resulting buffer, NULL if unsuccessful. The passed in size will get page aligned, if it isn't already.

## Name

`relay_create_buf` — allocate and initialize a channel buffer

## Synopsis

```
struct rchan_buf * relay_create_buf (struct rchan * chan);
```

## Arguments

*chan* the relay channel

## Description

Returns channel buffer if successful, NULL otherwise.

## Name

`relay_destroy_channel` — free the channel struct

## Synopsis

```
void relay_destroy_channel (struct kref * kref);
```

## Arguments

*kref* target kernel reference that contains the relay channel

## Description

Should only be called from `kref_put`.

## Name

`relay_destroy_buf` — destroy an `rchan_buf` struct and associated buffer

## Synopsis

```
void relay_destroy_buf (struct rchan_buf * buf);
```

## Arguments

*buf* the buffer struct

## Name

`relay_remove_buf` — remove a channel buffer

## Synopsis

```
void relay_remove_buf (struct kref * kref);
```

## Arguments

*kref* target kernel reference that contains the relay buffer

## Description

Removes the file from the filesystem, which also frees the `rchan_buf_struct` and the channel buffer. Should only be called from `kref_put`.

## Name

`relay_buf_empty` — boolean, is the channel buffer empty?

## Synopsis

```
int relay_buf_empty (struct rchan_buf * buf);
```

## Arguments

*buf*    channel buffer

## Description

Returns 1 if the buffer is empty, 0 otherwise.

## Name

wakeup\_readers — wake up readers waiting on a channel

## Synopsis

```
void wakeup_readers (unsigned long data);
```

## Arguments

*data* contains the channel buffer

## Description

This is the timer function used to defer reader waking.

## Name

`__relay_reset` — reset a channel buffer

## Synopsis

```
void __relay_reset (struct rchan_buf * buf, unsigned int init);
```

## Arguments

*buf*     the channel buffer

*init*    1 if this is a first-time initialization

## Description

See `relay_reset` for description of effect.



## Name

`relay_close_buf` — close a channel buffer

## Synopsis

```
void relay_close_buf (struct rchan_buf * buf);
```

## Arguments

*buf*    channel buffer

## Description

Marks the buffer finalized and restores the default callbacks. The channel buffer and channel buffer data structure are then freed automatically when the last reference is given up.

## Name

relay\_hotcpu\_callback — CPU hotplug callback

## Synopsis

```
int relay_hotcpu_callback (struct notifier_block * nb, unsigned long  
action, void * hcpu);
```

## Arguments

*nb*        notifier block

*action*   hotplug action to take

*hcpu*     CPU number

## Description

Returns the success/failure of the operation. (NOTIFY\_OK, NOTIFY\_BAD)

## Name

`relay_late_setup_files` — triggers file creation

## Synopsis

```
int relay_late_setup_files (struct rchan * chan, const char *  
base_filename, struct dentry * parent);
```

## Arguments

<i>chan</i>	channel to operate on
<i>base_filename</i>	base name of files to create
<i>parent</i>	dentry of parent directory, NULL for root directory

## Description

Returns 0 if successful, non-zero otherwise.

Use to setup files for a previously buffer-only channel. Useful to do early tracing in kernel, before VFS is up, for example.

## Name

relay\_file\_open — open file op for relay files

## Synopsis

```
int relay_file_open (struct inode * inode, struct file * filp);
```

## Arguments

*inode*    the inode

*filp*    the file

## Description

Increments the channel buffer refcount.

## Name

relay\_file\_mmap — mmap file op for relay files

## Synopsis

```
int relay_file_mmap (struct file * filp, struct vm_area_struct * vma);
```

## Arguments

*filp* the file

*vma* the vma describing what to map

## Description

Calls upon relay\_mmap\_buf to map the file into user space.

## Name

relay\_file\_poll — poll file op for relay files

## Synopsis

```
unsigned int relay_file_poll (struct file * filp, poll_table * wait);
```

## Arguments

*filp* the file

*wait* poll table

## Description

Poll implementation.

## Name

`relay_file_release` — release file op for relay files

## Synopsis

```
int relay_file_release (struct inode * inode, struct file * filp);
```

## Arguments

*inode*    the inode

*filp*    the file

## Description

Decrements the channel refcount, as the filesystem is no longer using it.

## Name

`relay_file_read_subbuf_avail` — return bytes available in sub-buffer

## Synopsis

```
size_t relay_file_read_subbuf_avail (size_t read_pos, struct rchan_buf
* buf);
```

## Arguments

*read\_pos*    file read position

*buf*            relay channel buffer



## Name

`relay_file_read_start_pos` — find the first available byte to read

## Synopsis

```
size_t relay_file_read_start_pos (size_t read_pos, struct rchan_buf *
buf);
```

## Arguments

*read\_pos*    file read position

*buf*            relay channel buffer

## Description

If the *read\_pos* is in the middle of padding, return the position of the first actually available byte, otherwise return the original value.

## Name

`relay_file_read_end_pos` — return the new read position

## Synopsis

```
size_t relay_file_read_end_pos (struct rchan_buf * buf, size_t read_pos,
size_t count);
```

## Arguments

<i>buf</i>	relay channel buffer
<i>read_pos</i>	file read position
<i>count</i>	number of bytes to be read

---

# **Chapter 8. Module Support**

## **Module Loading**

## Name

`__request_module` — try to load a kernel module

## Synopsis

```
int __request_module (bool wait, const char * fmt, ...);
```

## Arguments

*wait*    wait (or not) for the operation to complete

*fmt*     printf style format string for the name of the module @...: arguments as specified in the format string

*...*    variable arguments

## Description

Load a module using the user mode module loader. The function returns zero on success or a negative `errno` code or positive exit code from “`modprobe`” 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.

## Name

`call_usermodehelper_setup` — prepare to call a usermode helper

## Synopsis

```
struct subprocess_info * call_usermodehelper_setup (char * path, char **  
argv, char ** envp, gfp_t gfp_mask, int (*init) (struct subprocess_info  
*info, struct cred *new), void (*cleanup) (struct subprocess_info  
*info), void * data);
```

## Arguments

<i>path</i>	path to usermode executable
<i>argv</i>	arg vector for process
<i>envp</i>	environment for process
<i>gfp_mask</i>	gfp mask for memory allocation
<i>init</i>	an init function
<i>cleanup</i>	a cleanup function
<i>data</i>	arbitrary context sensitive data

## Description

Returns either `NULL` on allocation failure, or a `subprocess_info` structure. This should be passed to `call_usermodehelper_exec` to exec the process and free the structure.

The `init` function is used to customize the helper process prior to `exec`. A non-zero return code causes the process to error out, `exit`, and return the failure to the calling process

The `cleanup` function is just before the `subprocess_info` is about to be freed. This can be used for freeing the `argv` and `envp`. The Function must be runnable in either a process context or the context in which `call_usermodehelper_exec` is called.

## Name

`call_usermodehelper_exec` — start a usermode application

## Synopsis

```
int call_usermodehelper_exec (struct subprocess_info * sub_info, int  
wait);
```

## Arguments

*sub\_info*    information about the subprocessa

*wait*        wait for the application to finish and return status. when UMH\_NO\_WAIT don't wait at all, but you get no useful error back when the program couldn't be exec'ed. This makes it safe to call from interrupt context.

## Description

Runs a user-space application. The application is started asynchronously if `wait` is not set, and runs as a child of system workqueues. (ie. it runs with full root capabilities and optimized affinity).

## Name

`call_usermodehelper` — prepare and start a usermode application

## Synopsis

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

## Arguments

*path* path to usermode executable

*argv* arg vector for process

*envp* environment for process

*wait* wait for the application to finish and return status. when `UMH_NO_WAIT` don't wait at all, but you get no useful error back when the program couldn't be exec'ed. This makes it safe to call from interrupt context.

## Description

This function is the equivalent to use `call_usermodehelper_setup` and `call_usermodehelper_exec`.

## Inter Module support

Refer to the file `kernel/module.c` for more information.

---

# **Chapter 9. Hardware Interfaces**

## **Interrupt Handling**



## Name

`synchronize_hardirq` — wait for pending hard IRQ handlers (on other CPUs)

## Synopsis

```
bool synchronize_hardirq (unsigned int irq);
```

## Arguments

*irq* interrupt number to wait for

## Description

This function waits for any pending hard 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. It does not take associated threaded handlers into account.

Do not use this for shutdown scenarios where you must be sure that all parts (hardirq and threaded handler) have completed.

## Returns

false if a threaded handler is active.

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

## Name

synchronize\_irq — wait for pending IRQ handlers (on other CPUs)

## Synopsis

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

## Arguments

*irq*   interrupt number to wait for

## Description

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.

## Name

`irq_set_affinity_notifier` — control notification of IRQ affinity changes

## Synopsis

```
int    irq_set_affinity_notifier    (unsigned    int    irq,    struct
irq_affinity_notify * notify);
```

## Arguments

*irq* Interrupt for which to enable/disable notification

*notify* Context for notification, or NULL to disable notification. Function pointers must be initialised; the other fields will be initialised by this function.

## Description

Must be called in process context. Notification may only be enabled after the IRQ is allocated and must be disabled before the IRQ is freed using `free_irq`.

## Name

`irq_set_vcpu_affinity` — Set vcpu affinity for the interrupt

## Synopsis

```
int irq_set_vcpu_affinity (unsigned int irq, void * vcpu_info);
```

## Arguments

*irq*                    interrupt number to set affinity

*vcpu\_info*    vCPU specific data

## Description

This function uses the vCPU specific data to set the vCPU affinity for an irq. The vCPU specific data is passed from outside, such as KVM. One example code path is as below: KVM -> IOMMU -> `irq_set_vcpu_affinity`.

## Name

`disable_irq_nosync` — disable an irq without waiting

## Synopsis

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

## Arguments

*irq*    Interrupt to disable

## Description

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

This function may be called from IRQ context.

## Name

`disable_irq` — disable an irq and wait for completion

## Synopsis

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

## Arguments

*irq*    Interrupt to disable

## Description

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

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

## Name

`disable_hardirq` — disables an irq and waits for hardirq completion

## Synopsis

```
bool disable_hardirq (unsigned int irq);
```

## Arguments

*irq* Interrupt to disable

## Description

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

When used to optimistically disable an interrupt from atomic context the return value must be checked.

## Returns

false if a threaded handler is active.

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

## Name

`enable_irq` — enable handling of an irq

## Synopsis

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

## Arguments

*irq*    Interrupt to enable

## Description

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

This function may be called from IRQ context only when `desc->irq_data.chip->bus_lock` and `desc->chip->bus_sync_unlock` are NULL !



## Name

`irq_set_irq_wake` — control irq power management wakeup

## Synopsis

```
int irq_set_irq_wake (unsigned int irq, unsigned int on);
```

## Arguments

*irq*    interrupt to control

*on*    enable/disable power management wakeup

## Description

Enable/disable power management wakeup mode, which is disabled by default. Enables and disables must match, just as they match for non-wakeup mode support.

Wakeup mode lets this IRQ wake the system from sleep states like “suspend to RAM”.

## Name

`irq_wake_thread` — wake the irq thread for the action identified by `dev_id`

## Synopsis

```
void irq_wake_thread (unsigned int irq, void * dev_id);
```

## Arguments

*irq*      Interrupt line

*dev\_id*   Device identity for which the thread should be woken

## Name

`setup_irq` — setup an interrupt

## Synopsis

```
int setup_irq (unsigned int irq, struct irqaction * act);
```

## Arguments

*irq*    Interrupt line to setup

*act*    irqaction for the interrupt

## Description

Used to statically setup interrupts in the early boot process.

## Name

`remove_irq` — free an interrupt

## Synopsis

```
void remove_irq (unsigned int irq, struct irqaction * act);
```

## Arguments

*irq*    Interrupt line to free

*act*    irqaction for the interrupt

## Description

Used to remove interrupts statically setup by the early boot process.

## Name

`free_irq` — free an interrupt allocated with `request_irq`

## Synopsis

```
void free_irq (unsigned int irq, void * dev_id);
```

## Arguments

*irq*        Interrupt line to free

*dev\_id*    Device identity to free

## Description

Remove an interrupt handler. The handler is removed and if the interrupt line is no longer in use by any driver it is disabled. On a shared IRQ the caller must ensure the interrupt is disabled on the card it drives before calling this function. The function does not return until any executing interrupts for this IRQ have completed.

This function must not be called from interrupt context.

## Name

`request_threaded_irq` — allocate an interrupt line

## Synopsis

```
int request_threaded_irq (unsigned int irq, irq_handler_t handler,  
irq_handler_t thread_fn, unsigned long irqflags, const char * devname,  
void * dev_id);
```

## Arguments

<i>irq</i>	Interrupt line to allocate
<i>handler</i>	Function to be called when the IRQ occurs. Primary handler for threaded interrupts If NULL and <i>thread_fn</i> != NULL the default primary handler is installed
<i>thread_fn</i>	Function called from the irq handler thread If NULL, no irq thread is created
<i>irqflags</i>	Interrupt type flags
<i>devname</i>	An ascii name for the claiming device
<i>dev_id</i>	A cookie passed back to the handler function

## Description

This call allocates interrupt resources and enables the interrupt line and IRQ handling. From the point this call is made your handler function may be invoked. Since your handler function must clear any interrupt the board raises, you must take care both to initialise your hardware and to set up the interrupt handler in the right order.

If you want to set up a threaded irq handler for your device then you need to supply *handler* and *thread\_fn*. *handler* is still called in hard interrupt context and has to check whether the interrupt originates from the device. If yes it needs to disable the interrupt on the device and return `IRQ_WAKE_THREAD` which will wake up the handler thread and run *thread\_fn*. This split handler design is necessary to support shared interrupts.

*Dev\_id* must be globally unique. Normally the address of the device data structure is used as the cookie. Since the handler receives this value it makes sense to use it.

If your interrupt is shared you must pass a non NULL *dev\_id* as this is required when freeing the interrupt.

## Flags

`IRQF_SHARED` Interrupt is shared `IRQF_TRIGGER_*` Specify active edge(s) or level

## Name

`request_any_context_irq` — allocate an interrupt line

## Synopsis

```
int request_any_context_irq (unsigned int irq, irq_handler_t handler,
unsigned long flags, const char * name, void * dev_id);
```

## Arguments

<i>irq</i>	Interrupt line to allocate
<i>handler</i>	Function to be called when the IRQ occurs. Threaded handler for threaded interrupts.
<i>flags</i>	Interrupt type flags
<i>name</i>	An ascii name for the claiming device
<i>dev_id</i>	A cookie passed back to the handler function

## Description

This call allocates interrupt resources and enables the interrupt line and IRQ handling. It selects either a hardirq or threaded handling method depending on the context.

On failure, it returns a negative value. On success, it returns either `IRQC_IS_HARDIRQ` or `IRQC_IS_NESTED`.

## Name

`free_percpu_irq` — free an interrupt allocated with `request_percpu_irq`

## Synopsis

```
void free_percpu_irq (unsigned int irq, void __percpu * dev_id);
```

## Arguments

*irq*        Interrupt line to free

*dev\_id*    Device identity to free

## Description

Remove a percpu interrupt handler. The handler is removed, but the interrupt line is not disabled. This must be done on each CPU before calling this function. The function does not return until any executing interrupts for this IRQ have completed.

This function must not be called from interrupt context.



## Name

`request_percpu_irq` — allocate a percpu interrupt line

## Synopsis

```
int request_percpu_irq (unsigned int irq, irq_handler_t handler, const  
char * devname, void __percpu * dev_id);
```

## Arguments

<i>irq</i>	Interrupt line to allocate
<i>handler</i>	Function to be called when the IRQ occurs.
<i>devname</i>	An ascii name for the claiming device
<i>dev_id</i>	A percpu cookie passed back to the handler function

## Description

This call allocates interrupt resources and enables the interrupt on the local CPU. If the interrupt is supposed to be enabled on other CPUs, it has to be done on each CPU using `enable_percpu_irq`.

`Dev_id` must be globally unique. It is a per-cpu variable, and the handler gets called with the interrupted CPU's instance of that variable.

## Name

`irq_get_irqchip_state` — returns the irqchip state of a interrupt.

## Synopsis

```
int irq_get_irqchip_state (unsigned int irq, enum irqchip_irq_state
    which, bool * state);
```

## Arguments

*irq*      Interrupt line that is forwarded to a VM

*which*    One of `IRQCHIP_STATE_*` the caller wants to know about

*state*    a pointer to a boolean where the state is to be stored

## Description

This call snapshots the internal irqchip state of an interrupt, returning into *state* the bit corresponding to stage *which*

This function should be called with preemption disabled if the interrupt controller has per-cpu registers.

## Name

`irq_set_irqchip_state` — set the state of a forwarded interrupt.

## Synopsis

```
int irq_set_irqchip_state (unsigned int irq, enum irqchip_irq_state
which, bool val);
```

## Arguments

*irq*      Interrupt line that is forwarded to a VM

*which*    State to be restored (one of IRQCHIP\_STATE\_\*)

*val*      Value corresponding to *which*

## Description

This call sets the internal irqchip state of an interrupt, depending on the value of *which*.

This function should be called with preemption disabled if the interrupt controller has per-cpu registers.

## DMA Channels

## Name

`request_dma` — request and reserve a system DMA channel

## Synopsis

```
int request_dma (unsigned int dmanr, const char * device_id);
```

## Arguments

*dmanr*            DMA channel number

*device\_id*    reserving device ID string, used in /proc/dma

## Name

`free_dma` — free a reserved system DMA channel

## Synopsis

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

## Arguments

*dmanr* DMA channel number

## Resources Management

## Name

`request_resource_conflict` — request and reserve an I/O or memory resource

## Synopsis

```
struct resource * request_resource_conflict (struct resource * root,  
struct resource * new);
```

## Arguments

*root*    root resource descriptor

*new*    resource descriptor desired by caller

## Description

Returns 0 for success, conflict resource on error.

## Name

`region_intersects` — determine intersection of region with known resources

## Synopsis

```
int region_intersects (resource_size_t start, size_t size, const char
* name);
```

## Arguments

*start*    region start address

*size*     size of region

*name*     name of resource (in `iomem_resource`)

## Description

Check if the specified region partially overlaps or fully eclipses a resource identified by *name*. Return `REGION_DISJOINT` if the region does not overlap *name*, return `REGION_MIXED` if the region overlaps *type* and another resource, and return `REGION_INTERSECTS` if the region overlaps *type* and no other defined resource. Note, that `REGION_INTERSECTS` is also returned in the case when the specified region overlaps RAM and undefined memory holes.

`region_intersect` is used by memory remapping functions to ensure the user is not remapping RAM and is a vast speed up over walking through the resource table page by page.

## Name

`reallocate_resource` — allocate a slot in the resource tree given range & alignment. The resource will be relocated if the new size cannot be reallocated in the current location.

## Synopsis

```
int reallocate_resource (struct resource * root, struct resource * old,  
resource_size_t newsize, struct resource_constraint * constraint);
```

## Arguments

<i>root</i>	root resource descriptor
<i>old</i>	resource descriptor desired by caller
<i>newsize</i>	new size of the resource descriptor
<i>constraint</i>	the size and alignment constraints to be met.



## Name

`lookup_resource` — find an existing resource by a resource start address

## Synopsis

```
struct resource * lookup_resource (struct resource * root,  
resource_size_t start);
```

## Arguments

*root*    root resource descriptor

*start*   resource start address

## Description

Returns a pointer to the resource if found, NULL otherwise

## Name

`insert_resource_conflict` — Inserts resource in the resource tree

## Synopsis

```
struct resource * insert_resource_conflict (struct resource * parent,  
struct resource * new);
```

## Arguments

*parent*    parent of the new resource

*new*       new resource to insert

## Description

Returns 0 on success, conflict resource if the resource can't be inserted.

This function is equivalent to `request_resource_conflict` when no conflict happens. If a conflict happens, and the conflicting resources entirely fit within the range of the new resource, then the new resource is inserted and the conflicting resources become children of the new resource.

## Name

`insert_resource` — Inserts a resource in the resource tree

## Synopsis

```
int insert_resource (struct resource * parent, struct resource * new);
```

## Arguments

*parent*    parent of the new resource

*new*       new resource to insert

## Description

Returns 0 on success, -EBUSY if the resource can't be inserted.

## Name

`insert_resource_expand_to_fit` — Insert a resource into the resource tree

## Synopsis

```
void insert_resource_expand_to_fit (struct resource * root, struct  
resource * new);
```

## Arguments

*root*    root resource descriptor

*new*     new resource to insert

## Description

Insert a resource into the resource tree, possibly expanding it in order to make it encompass any conflicting resources.

## Name

`resource_alignment` — calculate resource's alignment

## Synopsis

```
resource_size_t resource_alignment (struct resource * res);
```

## Arguments

*res* resource pointer

## Description

Returns alignment on success, 0 (invalid alignment) on failure.

## Name

`release_mem_region_adjustable` — release a previously reserved memory region

## Synopsis

```
int  release_mem_region_adjustable (struct  resource  *  parent,
resource_size_t start, resource_size_t size);
```

## Arguments

*parent* parent resource descriptor

*start* resource start address

*size* resource region size

## Description

This interface is intended for memory hot-delete. The requested region is released from a currently busy memory resource. The requested region must either match exactly or fit into a single busy resource entry. In the latter case, the remaining resource is adjusted accordingly. Existing children of the busy memory resource must be immutable in the request.

## Note

- Additional release conditions, such as overlapping region, can be supported after they are confirmed as valid cases. - When a busy memory resource gets split into two entries, the code assumes that all children remain in the lower address entry for simplicity. Enhance this logic when necessary.

## Name

`request_resource` — request and reserve an I/O or memory resource

## Synopsis

```
int request_resource (struct resource * root, struct resource * new);
```

## Arguments

*root*    root resource descriptor

*new*    resource descriptor desired by caller

## Description

Returns 0 for success, negative error code on error.

## Name

`release_resource` — release a previously reserved resource

## Synopsis

```
int release_resource (struct resource * old);
```

## Arguments

*old* resource pointer



## Name

`allocate_resource` — allocate empty slot in the resource tree given range & alignment. The resource will be reallocated with a new size if it was already allocated

## Synopsis

```
int allocate_resource (struct resource * root, struct resource *  
new, resource_size_t size, resource_size_t min, resource_size_t max,  
resource_size_t align, resource_size_t (*alignf) (void *, const struct  
resource *, resource_size_t, resource_size_t), void * alignf_data);
```

## Arguments

<i>root</i>	root resource descriptor
<i>new</i>	resource descriptor desired by caller
<i>size</i>	requested resource region size
<i>min</i>	minimum boundary to allocate
<i>max</i>	maximum boundary to allocate
<i>align</i>	alignment requested, in bytes
<i>alignf</i>	alignment function, optional, called if not NULL
<i>alignf_data</i>	arbitrary data to pass to the <i>alignf</i> function

## Name

`adjust_resource` — modify a resource's start and size

## Synopsis

```
int adjust_resource (struct resource * res, resource_size_t start,  
resource_size_t size);
```

## Arguments

*res*      resource to modify

*start*    new start value

*size*     new size

## Description

Given an existing resource, change its start and size to match the arguments. Returns 0 on success, -EBUSY if it can't fit. Existing children of the resource are assumed to be immutable.

## Name

`__request_region` — create a new busy resource region

## Synopsis

```
struct resource * __request_region (struct resource * parent,  
resource_size_t start, resource_size_t n, const char * name, int flags);
```

## Arguments

<i>parent</i>	parent resource descriptor
<i>start</i>	resource start address
<i>n</i>	resource region size
<i>name</i>	reserving caller's ID string
<i>flags</i>	IO resource flags

## Name

`__release_region` — release a previously reserved resource region

## Synopsis

```
void __release_region (struct resource * parent, resource_size_t start,  
resource_size_t n);
```

## Arguments

*parent*    parent resource descriptor

*start*     resource start address

*n*         resource region size

## Description

The described resource region must match a currently busy region.

## Name

`devm_request_resource` — request and reserve an I/O or memory resource

## Synopsis

```
int devm_request_resource (struct device * dev, struct resource * root,  
struct resource * new);
```

## Arguments

*dev*     device for which to request the resource

*root*    root of the resource tree from which to request the resource

*new*     descriptor of the resource to request

## Description

This is a device-managed version of `request_resource`. There is usually no need to release resources requested by this function explicitly since that will be taken care of when the device is unbound from its driver. If for some reason the resource needs to be released explicitly, because of ordering issues for example, drivers must call `devm_release_resource` rather than the regular `release_resource`.

When a conflict is detected between any existing resources and the newly requested resource, an error message will be printed.

Returns 0 on success or a negative error code on failure.

## Name

`devm_release_resource` — release a previously requested resource

## Synopsis

```
void devm_release_resource (struct device * dev, struct resource * new);
```

## Arguments

*dev*    device for which to release the resource

*new*    descriptor of the resource to release

## Description

Releases a resource previously requested using `devm_request_resource`.

# MTRR Handling

## Name

`arch_phys_wc_add` — add a WC MTRR and handle errors if PAT is unavailable

## Synopsis

```
int arch_phys_wc_add (unsigned long base, unsigned long size);
```

## Arguments

*base*    Physical base address

*size*    Size of region

## Description

If PAT is available, this does nothing. If PAT is unavailable, it attempts to add a WC MTRR covering *size* bytes starting at *base* and logs an error if this fails.

The caller should provide a power of two size on an equivalent power of two boundary.

Drivers must store the return value to pass to `mtrr_del_wc_if_needed`, but drivers should not try to interpret that return value.

# PCI Support Library

## Name

`pci_bus_max_busnr` — returns maximum PCI bus number of given bus' children

## Synopsis

```
unsigned char pci_bus_max_busnr (struct pci_bus * bus);
```

## Arguments

*bus* pointer to PCI bus structure to search

## Description

Given a PCI bus, returns the highest PCI bus number present in the set including the given PCI bus and its list of child PCI buses.



## Name

`pci_find_capability` — query for devices' capabilities

## Synopsis

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

## Arguments

*dev*    PCI device to query

*cap*    capability code

## Description

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

<code>PCI_CAP_ID_PM</code>	Power Management	<code>PCI_CAP_ID_AGP</code>	Accelerated Graphics Port
<code>PCI_CAP_ID_VPD</code>	Vital Product Data	<code>PCI_CAP_ID_SLOTID</code>	Slot Identification
<code>PCI_CAP_ID_MSI</code>	Message Signalled Interrupts	<code>PCI_CAP_ID_CHSWP</code>	CompactPCI HotSwap
<code>PCI_CAP_ID_PCIX</code>	PCI-X	<code>PCI_CAP_ID_EXP</code>	PCI Express

## Name

`pci_bus_find_capability` — query for devices' capabilities

## Synopsis

```
int pci_bus_find_capability (struct pci_bus * bus, unsigned int devfn,  
int cap);
```

## Arguments

*bus*      the PCI bus to query

*devfn*    PCI device to query

*cap*      capability code

## Description

Like `pci_find_capability` but works for pci devices that do not have a `pci_dev` structure set up yet.

Returns the address of the requested capability structure within the device's PCI configuration space or 0 in case the device does not support it.

## Name

`pci_find_next_ext_capability` — Find an extended capability

## Synopsis

```
int pci_find_next_ext_capability (struct pci_dev * dev, int start, int  
cap);
```

## Arguments

*dev*     PCI device to query

*start*   address at which to start looking (0 to start at beginning of list)

*cap*     capability code

## Description

Returns the address of the next matching extended capability structure within the device's PCI configuration space or 0 if the device does not support it. Some capabilities can occur several times, e.g., the vendor-specific capability, and this provides a way to find them all.

## Name

`pci_find_ext_capability` — Find an extended capability

## Synopsis

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

## Arguments

*dev* PCI device to query

*cap* capability code

## Description

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

`PCI_EXT_CAP_ID_ERR` Advanced Error Reporting   `PCI_EXT_CAP_ID_VC` Virtual Channel  
`PCI_EXT_CAP_ID_DSN` Device Serial Number   `PCI_EXT_CAP_ID_PWR` Power Budgeting

## Name

`pci_find_next_ht_capability` — query a device's Hypertransport capabilities

## Synopsis

```
int pci_find_next_ht_capability (struct pci_dev * dev, int pos, int
ht_cap);
```

## Arguments

<i>dev</i>	PCI device to query
<i>pos</i>	Position from which to continue searching
<i>ht_cap</i>	Hypertransport capability code

## Description

To be used in conjunction with `pci_find_ht_capability` to search for all capabilities matching *ht\_cap*. *pos* should always be a value returned from `pci_find_ht_capability`.

NB. To be 100% safe against broken PCI devices, the caller should take steps to avoid an infinite loop.

## Name

`pci_find_ht_capability` — query a device's Hypertransport capabilities

## Synopsis

```
int pci_find_ht_capability (struct pci_dev * dev, int ht_cap);
```

## Arguments

*dev*        PCI device to query

*ht\_cap*    Hypertransport capability code

## Description

Tell if a device supports a given Hypertransport capability. Returns an address within the device's PCI configuration space or 0 in case the device does not support the request capability. The address points to the PCI capability, of type `PCI_CAP_ID_HT`, which has a Hypertransport capability matching *ht\_cap*.

## 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.

## Name

`pci_find_pcie_root_port` — return PCIe Root Port

## Synopsis

```
struct pci_dev * pci_find_pcie_root_port (struct pci_dev * dev);
```

## Arguments

*dev*    PCI device to query

## Description

Traverse up the parent chain and return the PCIe Root Port PCI Device for a given PCI Device.



## Name

`__pci_complete_power_transition` — Complete power transition of a PCI device

## Synopsis

```
int __pci_complete_power_transition (struct pci_dev * dev, pci_power_t
state);
```

## Arguments

*dev*     PCI device to handle.

*state*   State to put the device into.

## Description

This function should not be called directly by device drivers.

## Name

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

## Synopsis

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

## Arguments

*dev*     PCI device to handle.

*state*   PCI power state (D0, D1, D2, D3hot) to put the device into.

## Description

Transition a device to a new power state, using the platform firmware and/or the device's PCI PM registers.

## RETURN VALUE

-EINVAL if the requested state is invalid. -EIO if device does not support PCI PM or its PM capabilities register has a wrong version, or device doesn't support the requested state. 0 if device already is in the requested state. 0 if device's power state has been successfully changed.

## Name

`pci_choose_state` — Choose the power state of a PCI device

## Synopsis

```
pci_power_t pci_choose_state (struct pci_dev * dev, pm_message_t state);
```

## Arguments

*dev*     PCI device to be suspended

*state*   target sleep state for the whole system. This is the value that is passed to suspend function.

## Description

Returns PCI power state suitable for given device and given system message.

## Name

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

## Synopsis

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

## Arguments

*dev* - PCI device that we're dealing with

## Name

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

## Synopsis

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

## Arguments

*dev* - PCI device that we're dealing with

## Name

`pci_store_saved_state` — Allocate and return an opaque struct containing the device saved state.

## Synopsis

```
struct pci_saved_state * pci_store_saved_state (struct pci_dev * dev);
```

## Arguments

*dev*   PCI device that we're dealing with

## Description

Return NULL if no state or error.

## Name

`pci_load_saved_state` — Reload the provided save state into struct `pci_dev`.

## Synopsis

```
int pci_load_saved_state (struct pci_dev * dev, struct pci_saved_state  
* state);
```

## Arguments

*dev*     PCI device that we're dealing with

*state*   Saved state returned from `pci_store_saved_state`

## Name

`pci_load_and_free_saved_state` — Reload the save state pointed to by `state`, and free the memory allocated for it.

## Synopsis

```
int  pci_load_and_free_saved_state (struct pci_dev * dev, struct
pci_saved_state ** state);
```

## Arguments

*dev*     PCI device that we're dealing with

*state*   Pointer to saved state returned from `pci_store_saved_state`



## Name

`pci_reenable_device` — Resume abandoned device

## Synopsis

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

## Arguments

*dev* PCI device to be resumed

## Description

Note this function is a backend of `pci_default_resume` and is not supposed to be called by normal code, write proper resume handler and use it instead.

## Name

`pci_enable_device_io` — Initialize a device for use with IO space

## Synopsis

```
int pci_enable_device_io (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 resources. Wake up the device if it was suspended. Beware, this function can fail.

## Name

`pci_enable_device_mem` — Initialize a device for use with Memory space

## Synopsis

```
int pci_enable_device_mem (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 Memory resources. Wake up the device if it was suspended. Beware, this function can fail.

## 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.

Note we don't actually enable the device many times if we call this function repeatedly (we just increment the count).

## Name

`pcim_enable_device` — Managed `pci_enable_device`

## Synopsis

```
int pcim_enable_device (struct pci_dev * pdev);
```

## Arguments

*pdev*    PCI device to be initialized

## Description

Managed `pci_enable_device`.

## Name

`pcim_pin_device` — Pin managed PCI device

## Synopsis

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

## Arguments

*pdev*    PCI device to pin

## Description

Pin managed PCI device *pdev*. Pinned device won't be disabled on driver detach. *pdev* must have been enabled with `pcim_enable_device`.

## Name

`pci_disable_device` — Disable PCI device after use

## Synopsis

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

## Arguments

*dev* PCI device to be disabled

## Description

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

Note we don't actually disable the device until all callers of `pci_enable_device` have called `pci_disable_device`.

## Name

`pci_set_pcie_reset_state` — set reset state for device `dev`

## Synopsis

```
int    pci_set_pcie_reset_state    (struct    pci_dev    *    dev,    enum
pcie_reset_state    state);
```

## Arguments

*dev*     the PCIe device reset

*state*   Reset state to enter into

## Description

Sets the PCI reset state for the device.



## Name

`pci_pme_capable` — check the capability of PCI device to generate PME#

## Synopsis

```
bool pci_pme_capable (struct pci_dev * dev, pci_power_t state);
```

## Arguments

*dev*     PCI device to handle.

*state*   PCI state from which device will issue PME#.

## Name

`pci_pme_active` — enable or disable PCI device's PME# function

## Synopsis

```
void pci_pme_active (struct pci_dev * dev, bool enable);
```

## Arguments

*dev*        PCI device to handle.

*enable*    'true' to enable PME# generation; 'false' to disable it.

## Description

The caller must verify that the device is capable of generating PME# before calling this function with *enable* equal to 'true'.

## Name

`__pci_enable_wake` — enable PCI device as wakeup event source

## Synopsis

```
int __pci_enable_wake (struct pci_dev * dev, pci_power_t state, bool
runtime, bool enable);
```

## Arguments

<i>dev</i>	PCI device affected
<i>state</i>	PCI state from which device will issue wakeup events
<i>runtime</i>	True if the events are to be generated at run time
<i>enable</i>	True to enable event generation; false to disable

## Description

This enables the device as a wakeup event source, or disables it. When such events involves platform-specific hooks, those hooks are called automatically by this routine.

Devices with legacy power management (no standard PCI PM capabilities) always require such platform hooks.

## RETURN VALUE

0 is returned on success -EINVAL is returned if device is not supposed to wake up the system Error code depending on the platform is returned if both the platform and the native mechanism fail to enable the generation of wake-up events

## Name

`pci_wake_from_d3` — enable/disable device to wake up from D3\_hot or D3\_cold

## Synopsis

```
int pci_wake_from_d3 (struct pci_dev * dev, bool enable);
```

## Arguments

*dev*        PCI device to prepare

*enable*    True to enable wake-up event generation; false to disable

## Description

Many drivers want the device to wake up the system from D3\_hot or D3\_cold and this function allows them to set that up cleanly - `pci_enable_wake` should not be called twice in a row to enable wake-up due to PCI PM vs ACPI ordering constraints.

This function only returns error code if the device is not capable of generating PME# from both D3\_hot and D3\_cold, and the platform is unable to enable wake-up power for it.

## Name

`pci_prepare_to_sleep` — prepare PCI device for system-wide transition into a sleep state

## Synopsis

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

## Arguments

*dev* Device to handle.

## Description

Choose the power state appropriate for the device depending on whether it can wake up the system and/or is power manageable by the platform (PCI\_D3hot is the default) and put the device into that state.

## Name

`pci_back_from_sleep` — turn PCI device on during system-wide transition into working state

## Synopsis

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

## Arguments

*dev* Device to handle.

## Description

Disable device's system wake-up capability and put it into D0.

## Name

`pci_dev_run_wake` — Check if device can generate run-time wake-up events.

## Synopsis

```
bool pci_dev_run_wake (struct pci_dev * dev);
```

## Arguments

*dev* Device to check.

## Description

Return true if the device itself is capable of generating wake-up events (through the platform or using the native PCIe PME) or if the device supports PME and one of its upstream bridges can generate wake-up events.

## Name

`pci_common_swizzle` — swizzle INTx all the way to root bridge

## Synopsis

```
u8 pci_common_swizzle (struct pci_dev * dev, u8 * pinp);
```

## Arguments

*dev*     the PCI device

*pinp*    pointer to the INTx pin value (1=INTA, 2=INTB, 3=INTD, 4=INTD)

## Description

Perform INTx swizzling for a device. This traverses through all PCI-to-PCI bridges all the way up to a PCI root bus.



## Name

`pci_release_region` — Release a PCI bar

## Synopsis

```
void pci_release_region (struct pci_dev * pdev, int bar);
```

## Arguments

*pdev*    PCI device whose resources were previously reserved by `pci_request_region`

*bar*     BAR to release

## Description

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

## Name

`pci_request_region` — Reserve PCI I/O and memory resource

## Synopsis

```
int pci_request_region (struct pci_dev * pdev, int bar, const char *  
res_name);
```

## Arguments

*pdev*            PCI device whose resources are to be reserved

*bar*            BAR to be reserved

*res\_name*      Name to be associated with resource

## Description

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

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

## Name

`pci_request_region_exclusive` — Reserved PCI I/O and memory resource

## Synopsis

```
int pci_request_region_exclusive (struct pci_dev * pdev, int bar, const
char * res_name);
```

## Arguments

*pdev*            PCI device whose resources are to be reserved

*bar*            BAR to be reserved

*res\_name*      Name to be associated with resource.

## Description

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

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

The key difference that `_exclusive` makes it that userspace is explicitly not allowed to map the resource via `/dev/mem` or `sysfs`.

## Name

`pci_release_selected_regions` — Release selected PCI I/O and memory resources

## Synopsis

```
void pci_release_selected_regions (struct pci_dev * pdev, int bars);
```

## Arguments

*pdev* PCI device whose resources were previously reserved

*bars* Bitmask of BARs to be released

## Description

Release selected PCI I/O and memory resources previously reserved. Call this function only after all use of the PCI regions has ceased.

## Name

`pci_request_selected_regions` — Reserve selected PCI I/O and memory resources

## Synopsis

```
int pci_request_selected_regions (struct pci_dev * pdev, int bars, const  
char * res_name);
```

## Arguments

<i>pdev</i>	PCI device whose resources are to be reserved
<i>bars</i>	Bitmask of BARs to be requested
<i>res_name</i>	Name to be associated with resource

## Name

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

## Synopsis

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

## Arguments

*pdev* PCI device whose resources were previously reserved by `pci_request_regions`

## Description

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

## Name

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

## Synopsis

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

## Arguments

*pdev*            PCI device whose resources are to be reserved

*res\_name*       Name to be associated with resource.

## Description

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

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

## Name

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

## Synopsis

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

## Arguments

*pdev*            PCI device whose resources are to be reserved

*res\_name*       Name to be associated with resource.

## Description

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

`pci_request_regions_exclusive` will mark the region so that `/dev/mem` and the `sysfs` MMIO access will not be allowed.

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



## Name

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

## Synopsis

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

## Arguments

*dev* the PCI device to enable

## Description

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

## Name

`pci_clear_master` — disables bus-mastering for device `dev`

## Synopsis

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

## Arguments

*dev* the PCI device to disable

## Name

`pci_set_cacheline_size` — ensure the `CACHE_LINE_SIZE` register is programmed

## Synopsis

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

## Arguments

*dev* the PCI device for which MWI is to be enabled

## Description

Helper function for `pci_set_mwi`. Originally copied from `drivers/net/acenic.c`. Copyright 1998-2001 by Jes Sorensen, <*jestrained-monkey.org*>.

## RETURNS

An appropriate `-ERRNO` error value on error, or zero for success.

## Name

`pci_set_mwi` — enables memory-write-invalidate PCI transaction

## Synopsis

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

## Arguments

*dev* the PCI device for which MWI is enabled

## Description

Enables the Memory-Write-Invalidate transaction in `PCI_COMMAND`.

## RETURNS

An appropriate `-ERRNO` error value on error, or zero for success.

## Name

`pci_try_set_mwi` — enables memory-write-invalidate PCI transaction

## Synopsis

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

## Arguments

*dev* the PCI device for which MWI is enabled

## Description

Enables the Memory-Write-Invalidate transaction in `PCI_COMMAND`. Callers are not required to check the return value.

## RETURNS

An appropriate `-ERRNO` error value on error, or zero for success.

## Name

`pci_clear_mwi` — disables Memory-Write-Invalidate for device `dev`

## Synopsis

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

## Arguments

`dev` the PCI device to disable

## Description

Disables PCI Memory-Write-Invalidate transaction on the device

## Name

`pci_intx` — enables/disables PCI INTx for device `dev`

## Synopsis

```
void pci_intx (struct pci_dev * pdev, int enable);
```

## Arguments

*pdev*      the PCI device to operate on

*enable*    boolean: whether to enable or disable PCI INTx

## Description

Enables/disables PCI INTx for device `dev`

## Name

`pci_intx_mask_supported` — probe for INTx masking support

## Synopsis

```
bool pci_intx_mask_supported (struct pci_dev * dev);
```

## Arguments

*dev* the PCI device to operate on

## Description

Check if the device *dev* support INTx masking via the config space command word.



## Name

`pci_check_and_mask_intx` — mask INTx on pending interrupt

## Synopsis

```
bool pci_check_and_mask_intx (struct pci_dev * dev);
```

## Arguments

*dev* the PCI device to operate on

## Description

Check if the device *dev* has its INTx line asserted, mask it and return true in that case. False is returned if not interrupt was pending.

## Name

`pci_check_and_unmask_intx` — unmask INTx if no interrupt is pending

## Synopsis

```
bool pci_check_and_unmask_intx (struct pci_dev * dev);
```

## Arguments

*dev* the PCI device to operate on

## Description

Check if the device *dev* has its INTx line asserted, unmask it if not and return true. False is returned and the mask remains active if there was still an interrupt pending.

## Name

`pci_wait_for_pending_transaction` — waits for pending transaction

## Synopsis

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

## Arguments

*dev* the PCI device to operate on

## Description

Return 0 if transaction is pending 1 otherwise.

## Name

`pci_reset_bridge_secondary_bus` — Reset the secondary bus on a PCI bridge.

## Synopsis

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

## Arguments

*dev*    Bridge device

## Description

Use the bridge control register to assert reset on the secondary bus. Devices on the secondary bus are left in power-on state.

## Name

`__pci_reset_function` — reset a PCI device function

## Synopsis

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

## Arguments

*dev*   PCI device to reset

## Description

Some devices allow an individual function to be reset without affecting other functions in the same device. The PCI device must be responsive to PCI config space in order to use this function.

The device function is presumed to be unused when this function is called. Resetting the device will make the contents of PCI configuration space random, so any caller of this must be prepared to reinitialise the device including MSI, bus mastering, BARs, decoding IO and memory spaces, etc.

Returns 0 if the device function was successfully reset or negative if the device doesn't support resetting a single function.

## Name

`__pci_reset_function_locked` — reset a PCI device function while holding the *dev* mutex lock.

## Synopsis

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

## Arguments

*dev*   PCI device to reset

## Description

Some devices allow an individual function to be reset without affecting other functions in the same device. The PCI device must be responsive to PCI config space in order to use this function.

The device function is presumed to be unused and the caller is holding the device mutex lock when this function is called. Resetting the device will make the contents of PCI configuration space random, so any caller of this must be prepared to reinitialise the device including MSI, bus mastering, BARs, decoding IO and memory spaces, etc.

Returns 0 if the device function was successfully reset or negative if the device doesn't support resetting a single function.

## Name

`pci_reset_function` — quiesce and reset a PCI device function

## Synopsis

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

## Arguments

*dev*   PCI device to reset

## Description

Some devices allow an individual function to be reset without affecting other functions in the same device. The PCI device must be responsive to PCI config space in order to use this function.

This function does not just reset the PCI portion of a device, but clears all the state associated with the device. This function differs from `__pci_reset_function` in that it saves and restores device state over the reset.

Returns 0 if the device function was successfully reset or negative if the device doesn't support resetting a single function.

## Name

`pci_try_reset_function` — quiesce and reset a PCI device function

## Synopsis

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

## Arguments

*dev*    PCI device to reset

## Description

Same as above, except return -EAGAIN if unable to lock device.



## Name

`pci_probe_reset_slot` — probe whether a PCI slot can be reset

## Synopsis

```
int pci_probe_reset_slot (struct pci_slot * slot);
```

## Arguments

*slot* PCI slot to probe

## Description

Return 0 if slot can be reset, negative if a slot reset is not supported.

## Name

`pci_reset_slot` — reset a PCI slot

## Synopsis

```
int pci_reset_slot (struct pci_slot * slot);
```

## Arguments

*slot*    PCI slot to reset

## Description

A PCI bus may host multiple slots, each slot may support a reset mechanism independent of other slots. For instance, some slots may support slot power control. In the case of a 1:1 bus to slot architecture, this function may wrap the bus reset to avoid spurious slot related events such as hotplug. Generally a slot reset should be attempted before a bus reset. All of the function of the slot and any subordinate buses behind the slot are reset through this function. PCI config space of all devices in the slot and behind the slot is saved before and restored after reset.

Return 0 on success, non-zero on error.

## Name

`pci_try_reset_slot` — Try to reset a PCI slot

## Synopsis

```
int pci_try_reset_slot (struct pci_slot * slot);
```

## Arguments

*slot*   PCI slot to reset

## Description

Same as above except return -EAGAIN if the slot cannot be locked

## Name

`pci_probe_reset_bus` — probe whether a PCI bus can be reset

## Synopsis

```
int pci_probe_reset_bus (struct pci_bus * bus);
```

## Arguments

*bus*    PCI bus to probe

## Description

Return 0 if bus can be reset, negative if a bus reset is not supported.

## Name

`pci_reset_bus` — reset a PCI bus

## Synopsis

```
int pci_reset_bus (struct pci_bus * bus);
```

## Arguments

*bus*    top level PCI bus to reset

## Description

Do a bus reset on the given bus and any subordinate buses, saving and restoring state of all devices.

Return 0 on success, non-zero on error.

## Name

`pci_try_reset_bus` — Try to reset a PCI bus

## Synopsis

```
int pci_try_reset_bus (struct pci_bus * bus);
```

## Arguments

*bus*    top level PCI bus to reset

## Description

Same as above except return -EAGAIN if the bus cannot be locked

## Name

`pcix_get_max_mmrbc` — get PCI-X maximum designed memory read byte count

## Synopsis

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

## Arguments

*dev*    PCI device to query

## Returns mmrbc

maximum designed memory read count in bytes or appropriate error value.

## Name

`pcix_get_mmrbc` — get PCI-X maximum memory read byte count

## Synopsis

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

## Arguments

*dev*    PCI device to query

## Returns `mmrbc`

maximum memory read count in bytes or appropriate error value.



## Name

`pcix_set_mmrbc` — set PCI-X maximum memory read byte count

## Synopsis

```
int pcix_set_mmrbc (struct pci_dev * dev, int mmrbc);
```

## Arguments

*dev*      PCI device to query

*mmrbc*    maximum memory read count in bytes valid values are 512, 1024, 2048, 4096

## Description

If possible sets maximum memory read byte count, some bridges have erratas that prevent this.

## Name

`pcie_get_readrq` — get PCI Express read request size

## Synopsis

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

## Arguments

*dev*    PCI device to query

## Description

Returns maximum memory read request in bytes or appropriate error value.

## Name

`pcie_set_readrq` — set PCI Express maximum memory read request

## Synopsis

```
int pcie_set_readrq (struct pci_dev * dev, int rq);
```

## Arguments

*dev*    PCI device to query

*rq*     maximum memory read count in bytes valid values are 128, 256, 512, 1024, 2048, 4096

## Description

If possible sets maximum memory read request in bytes

## Name

`pcie_get_mps` — get PCI Express maximum payload size

## Synopsis

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

## Arguments

*dev*    PCI device to query

## Description

Returns maximum payload size in bytes

## Name

`pcie_set_mps` — set PCI Express maximum payload size

## Synopsis

```
int pcie_set_mps (struct pci_dev * dev, int mps);
```

## Arguments

*dev*    PCI device to query

*mps*    maximum payload size in bytes valid values are 128, 256, 512, 1024, 2048, 4096

## Description

If possible sets maximum payload size

## Name

`pcie_get_minimum_link` — determine minimum link settings of a PCI device

## Synopsis

```
int pcie_get_minimum_link (struct pci_dev * dev, enum pci_bus_speed *  
speed, enum pcie_link_width * width);
```

## Arguments

*dev*      PCI device to query

*speed*    storage for minimum speed

*width*    storage for minimum width

## Description

This function will walk up the PCI device chain and determine the minimum link width and speed of the device.

## Name

`pci_select_bars` — Make BAR mask from the type of resource

## Synopsis

```
int pci_select_bars (struct pci_dev * dev, unsigned long flags);
```

## Arguments

*dev*      the PCI device for which BAR mask is made

*flags*    resource type mask to be selected

## Description

This helper routine makes bar mask from the type of resource.

## Name

`pci_add_dynid` — add a new PCI device ID to this driver and re-probe devices

## Synopsis

```
int pci_add_dynid (struct pci_driver * drv, unsigned int vendor, unsigned
int device, unsigned int subvendor, unsigned int subdevice, unsigned
int class, unsigned int class_mask, unsigned long driver_data);
```

## Arguments

<i>drv</i>	target pci driver
<i>vendor</i>	PCI vendor ID
<i>device</i>	PCI device ID
<i>subvendor</i>	PCI subvendor ID
<i>subdevice</i>	PCI subdevice ID
<i>class</i>	PCI class
<i>class_mask</i>	PCI class mask
<i>driver_data</i>	private driver data

## Description

Adds a new dynamic pci device ID to this driver and causes the driver to probe for all devices again. *drv* must have been registered prior to calling this function.

## CONTEXT

Does GFP\_KERNEL allocation.

## RETURNS

0 on success, -errno on failure.



## Name

`pci_match_id` — See if a pci device matches a given `pci_id` table

## Synopsis

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

## Arguments

*ids*    array of PCI device id structures to search in

*dev*    the PCI device structure to match against.

## Description

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

Deprecated, don't use this as it will not catch any dynamic ids that a driver might want to check for.

## Name

`__pci_register_driver` — register a new pci driver

## Synopsis

```
int __pci_register_driver (struct pci_driver * drv, struct module *  
owner, const char * mod_name);
```

## Arguments

*drv*            the driver structure to register

*owner*        owner module of drv

*mod\_name*    module name string

## Description

Adds the driver structure to the list of registered drivers. Returns a negative value on error, otherwise 0. If no error occurred, the driver remains registered even if no device was claimed during registration.

## Name

`pci_unregister_driver` — unregister a pci driver

## Synopsis

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

## Arguments

*drv* the driver structure to unregister

## Description

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

## Name

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

## Synopsis

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

## Arguments

*dev* the device to query

## Description

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

## Name

`pci_dev_get` — increments the reference count of the pci device structure

## Synopsis

```
struct pci_dev * pci_dev_get (struct pci_dev * dev);
```

## Arguments

*dev* the device being referenced

## Description

Each live reference to a device should be refcounted.

Drivers for PCI devices should normally record such references in their `probe` methods, when they bind to a device, and release them by calling `pci_dev_put`, in their `disconnect` methods.

A pointer to the device with the incremented reference counter is returned.

## Name

`pci_dev_put` — release a use of the pci device structure

## Synopsis

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

## Arguments

*dev* device that's been disconnected

## Description

Must be called when a user of a device is finished with it. When the last user of the device calls this function, the memory of the device is freed.

## Name

`pci_stop_and_remove_bus_device` — remove a PCI device and any children

## Synopsis

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

## Arguments

*dev* the device to remove

## Description

Remove a PCI device from the device lists, informing the drivers that the device has been removed. We also remove any subordinate buses and children in a depth-first manner.

For each device we remove, delete the device structure from the device lists, remove the `/proc` entry, and notify userspace (`/sbin/hotplug`).

## Name

`pci_find_bus` — locate PCI bus from a given domain and bus number

## Synopsis

```
struct pci_bus * pci_find_bus (int domain, int busnr);
```

## Arguments

*domain*    number of PCI domain to search

*busnr*     number of desired PCI bus

## Description

Given a PCI bus number and domain number, the desired PCI bus is located in the global list of PCI buses. If the bus is found, a pointer to its data structure is returned. If no bus is found, NULL is returned.



## Name

`pci_find_next_bus` — begin or continue searching for a PCI bus

## Synopsis

```
struct pci_bus * pci_find_next_bus (const struct pci_bus * from);
```

## Arguments

*from* Previous PCI bus found, or NULL for new search.

## Description

Iterates through the list of known PCI buses. A new search is initiated by passing NULL as the *from* argument. Otherwise if *from* is not NULL, searches continue from next device on the global list.

## Name

`pci_get_slot` — locate PCI device for a given PCI slot

## Synopsis

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

## Arguments

*bus*      PCI bus on which desired PCI device resides

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

## Description

Given a PCI bus and slot/function number, the desired PCI device is located in the list of PCI devices. If the device is found, its reference count is increased and this function returns a pointer to its data structure. The caller must decrement the reference count by calling `pci_dev_put`. If no device is found, `NULL` is returned.

## Name

`pci_get_domain_bus_and_slot` — locate PCI device for a given PCI domain (segment), bus, and slot

## Synopsis

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

## Arguments

*domain*    PCI domain/segment on which the PCI device resides.

*bus*        PCI bus on which desired PCI device resides

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

## Description

Given a PCI domain, bus, and slot/function number, the desired PCI device is located in the list of PCI devices. If the device is found, its reference count is increased and this function returns a pointer to its data structure. The caller must decrement the reference count by calling `pci_dev_put`. If no device is found, `NULL` is returned.

## Name

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

## Synopsis

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

## Arguments

<i>vendor</i>	PCI vendor id to match, or <code>PCI_ANY_ID</code> to match all vendor ids
<i>device</i>	PCI device id to match, or <code>PCI_ANY_ID</code> to match all device ids
<i>ss_vendor</i>	PCI subsystem vendor id to match, or <code>PCI_ANY_ID</code> to match all vendor ids
<i>ss_device</i>	PCI subsystem device id to match, or <code>PCI_ANY_ID</code> to match all device ids
<i>from</i>	Previous PCI device found in search, or <code>NULL</code> for new search.

## Description

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

## Name

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

## Synopsis

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

## Arguments

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

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

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

## Description

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

## Name

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

## Synopsis

```
struct pci_dev * pci_get_class (unsigned int class, 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*, the reference count to the device is incremented and a pointer to its device structure is returned. Otherwise, NULL is returned. A new search is initiated by passing NULL as the *from* argument. Otherwise if *from* is not NULL, searches continue from next device on the global list. The reference count for *from* is always decremented if it is not NULL.

## Name

`pci_dev_present` — Returns 1 if device matching the device list is present, 0 if not.

## Synopsis

```
int pci_dev_present (const struct pci_device_id * ids);
```

## Arguments

*ids* A pointer to a null terminated list of struct `pci_device_id` structures that describe the type of PCI device the caller is trying to find.

## Obvious fact

You do not have a reference to any device that might be found by this function, so if that device is removed from the system right after this function is finished, the value will be stale. Use this function to find devices that are usually built into a system, or for a general hint as to if another device happens to be present at this specific moment in time.

## Name

`pci_msi_vec_count` — Return the number of MSI vectors a device can send

## Synopsis

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

## Arguments

*dev* device to report about

## Description

This function returns the number of MSI vectors a device requested via Multiple Message Capable register. It returns a negative `errno` if the device is not capable sending MSI interrupts. Otherwise, the call succeeds and returns a power of two, up to a maximum of  $2^5$  (32), according to the MSI specification.



## Name

`pci_msix_vec_count` — return the number of device's MSI-X table entries

## Synopsis

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

## Arguments

*dev* pointer to the `pci_dev` data structure of MSI-X device function This function returns the number of device's MSI-X table entries and therefore the number of MSI-X vectors device is capable of sending. It returns a negative `errno` if the device is not capable of sending MSI-X interrupts.

## Name

`pci_enable_msix` — configure device's MSI-X capability structure

## Synopsis

```
int pci_enable_msix (struct pci_dev * dev, struct msix_entry * entries,  
int nvec);
```

## Arguments

*dev*            pointer to the `pci_dev` data structure of MSI-X device function

*entries*       pointer to an array of MSI-X entries

*nvec*           number of MSI-X irqs requested for allocation by device driver

## Description

Setup the MSI-X capability structure of device function with the number of requested irqs upon its software driver call to request for MSI-X mode enabled on its hardware device function. A return of zero indicates the successful configuration of MSI-X capability structure with new allocated MSI-X irqs. A return of  $< 0$  indicates a failure. Or a return of  $> 0$  indicates that driver request is exceeding the number of irqs or MSI-X vectors available. Driver should use the returned value to re-send its request.

## Name

`pci_msi_enabled` — is MSI enabled?

## Synopsis

```
int pci_msi_enabled ( void );
```

## Arguments

*void* no arguments

## Description

Returns true if MSI has not been disabled by the command-line option `pci=noms`.

## Name

`pci_enable_msi_range` — configure device's MSI capability structure

## Synopsis

```
int pci_enable_msi_range (struct pci_dev * dev, int minvec, int maxvec);
```

## Arguments

*dev*        device to configure

*minvec*    minimal number of interrupts to configure

*maxvec*    maximum number of interrupts to configure

## Description

This function tries to allocate a maximum possible number of interrupts in a range between *minvec* and *maxvec*. It returns a negative `errno` if an error occurs. If it succeeds, it returns the actual number of interrupts allocated and updates the *dev*'s `irq` member to the lowest new interrupt number; the other interrupt numbers allocated to this device are consecutive.

## Name

`pci_enable_msix_range` — configure device's MSI-X capability structure

## Synopsis

```
int pci_enable_msix_range (struct pci_dev * dev, struct msix_entry *  
entries, int minvec, int maxvec);
```

## Arguments

*dev*            pointer to the `pci_dev` data structure of MSI-X device function

*entries*       pointer to an array of MSI-X entries

*minvec*        minimum number of MSI-X irqs requested

*maxvec*        maximum number of MSI-X irqs requested

## Description

Setup the MSI-X capability structure of device function with a maximum possible number of interrupts in the range between *minvec* and *maxvec* upon its software driver call to request for MSI-X mode enabled on its hardware device function. It returns a negative `errno` if an error occurs. If it succeeds, it returns the actual number of interrupts allocated and indicates the successful configuration of MSI-X capability structure with new allocated MSI-X interrupts.

## Name

`pci_bus_alloc_resource` — allocate a resource from a parent bus

## Synopsis

```
int pci_bus_alloc_resource (struct pci_bus * bus, struct resource *  
res, resource_size_t size, resource_size_t align, resource_size_t min,  
unsigned long type_mask, resource_size_t (*alignf) (void *, const struct  
resource *, resource_size_t, resource_size_t), void * alignf_data);
```

## Arguments

<i>bus</i>	PCI bus
<i>res</i>	resource to allocate
<i>size</i>	size of resource to allocate
<i>align</i>	alignment of resource to allocate
<i>min</i>	minimum /proc/iomem address to allocate
<i>type_mask</i>	IORESOURCE_* type flags
<i>alignf</i>	resource alignment function
<i>alignf_data</i>	data argument for resource alignment function

## Description

Given the PCI bus a device resides on, the size, minimum address, alignment and type, try to find an acceptable resource allocation for a specific device resource.

## Name

`pci_bus_add_device` — start driver for a single device

## Synopsis

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

## Arguments

*dev* device to add

## Description

This adds add sysfs entries and start device drivers

## Name

`pci_bus_add_devices` — start driver for PCI devices

## Synopsis

```
void pci_bus_add_devices (const struct pci_bus * bus);
```

## Arguments

*bus*    bus to check for new devices

## Description

Start driver for PCI devices and add some sysfs entries.



## Name

`pci_bus_set_ops` — Set raw operations of pci bus

## Synopsis

```
struct pci_ops * pci_bus_set_ops (struct pci_bus * bus, struct pci_ops  
* ops);
```

## Arguments

*bus* pci bus struct

*ops* new raw operations

## Description

Return previous raw operations

## Name

`pci_read_vpd` — Read one entry from Vital Product Data

## Synopsis

```
ssize_t pci_read_vpd (struct pci_dev * dev, loff_t pos, size_t count,  
void * buf);
```

## Arguments

<i>dev</i>	pci device struct
<i>pos</i>	offset in vpd space
<i>count</i>	number of bytes to read
<i>buf</i>	pointer to where to store result

## Name

`pci_write_vpd` — Write entry to Vital Product Data

## Synopsis

```
ssize_t pci_write_vpd (struct pci_dev * dev, loff_t pos, size_t count,  
const void * buf);
```

## Arguments

<i>dev</i>	pci device struct
<i>pos</i>	offset in vpd space
<i>count</i>	number of bytes to write
<i>buf</i>	buffer containing write data

## Name

`pci_cfg_access_lock` — Lock PCI config reads/writes

## Synopsis

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

## Arguments

*dev* pci device struct

## Description

When access is locked, any userspace reads or writes to config space and concurrent lock requests will sleep until access is allowed via `pci_cfg_access_unlocked` again.

## Name

`pci_cfg_access_trylock` — try to lock PCI config reads/writes

## Synopsis

```
bool pci_cfg_access_trylock (struct pci_dev * dev);
```

## Arguments

*dev* pci device struct

## Description

Same as `pci_cfg_access_lock`, but will return 0 if access is already locked, 1 otherwise. This function can be used from atomic contexts.

## Name

`pci_cfg_access_unlock` — Unlock PCI config reads/writes

## Synopsis

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

## Arguments

*dev* pci device struct

## Description

This function allows PCI config accesses to resume.

## Name

`pci_lost_interrupt` — reports a lost PCI interrupt

## Synopsis

```
enum pci_lost_interrupt_reason pci_lost_interrupt (struct pci_dev *  
pdev);
```

## Arguments

*pdev* device whose interrupt is lost

## Description

The primary function of this routine is to report a lost interrupt in a standard way which users can recognise (instead of blaming the driver).

## Returns

a suggestion for fixing it (although the driver is not required to act on this).

## Name

`__ht_create_irq` — create an irq and attach it to a device.

## Synopsis

```
int __ht_create_irq (struct pci_dev * dev, int idx, ht_irq_update_t *  
update);
```

## Arguments

*dev*        The hypertransport device to find the irq capability on.

*idx*        Which of the possible irqs to attach to.

*update*    Function to be called when changing the htirq message

## Description

The irq number of the new irq or a negative error value is returned.



## Name

`ht_create_irq` — create an irq and attach it to a device.

## Synopsis

```
int ht_create_irq (struct pci_dev * dev, int idx);
```

## Arguments

*dev*    The hypertransport device to find the irq capability on.

*idx*    Which of the possible irqs to attach to.

## Description

`ht_create_irq` needs to be called for all hypertransport devices that generate irqs.

The irq number of the new irq or a negative error value is returned.

## Name

`ht_destroy_irq` — destroy an irq created with `ht_create_irq`

## Synopsis

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

## Arguments

*irq*   irq to be destroyed

## Description

This reverses `ht_create_irq` removing the specified irq from existence. The irq should be free before this happens.

## Name

`pci_scan_slot` — scan a PCI slot on a bus for devices.

## Synopsis

```
int pci_scan_slot (struct pci_bus * bus, int devfn);
```

## Arguments

*bus*      PCI bus to scan

*devfn*   slot number to scan (must have zero function.)

## Description

Scan a PCI slot on the specified PCI bus for devices, adding discovered devices to the *bus*->devices list. New devices will not have `is_added` set.

Returns the number of new devices found.

## Name

`pci_rescan_bus` — scan a PCI bus for devices.

## Synopsis

```
unsigned int pci_rescan_bus (struct pci_bus * bus);
```

## Arguments

*bus*    PCI bus to scan

## Description

Scan a PCI bus and child buses for new devices, adds them, and enables them.

Returns the max number of subordinate bus discovered.

## Name

`pci_create_slot` — create or increment refcount for physical PCI slot

## Synopsis

```
struct pci_slot * pci_create_slot (struct pci_bus * parent, int slot_nr,  
const char * name, struct hotplug_slot * hotplug);
```

## Arguments

*parent*     struct pci\_bus of parent bridge

*slot\_nr*    PCI\_SLOT(pci\_dev->devfn) or -1 for placeholder

*name*       user visible string presented in /sys/bus/pci/slots/<name>

*hotplug*    set if caller is hotplug driver, NULL otherwise

## Description

PCI slots have first class attributes such as address, speed, width, and a struct pci\_slot is used to manage them. This interface will either return a new struct pci\_slot to the caller, or if the pci\_slot already exists, its refcount will be incremented.

Slots are uniquely identified by a *pci\_bus*, *slot\_nr* tuple.

There are known platforms with broken firmware that assign the same name to multiple slots. Workaround these broken platforms by renaming the slots on behalf of the caller. If firmware assigns name N to

## multiple slots

The first slot is assigned N The second slot is assigned N-1 The third slot is assigned N-2 etc.

## Placeholder slots

In most cases, *pci\_bus*, *slot\_nr* will be sufficient to uniquely identify a slot. There is one notable exception - pSeries (rpaphp), where the *slot\_nr* cannot be determined until a device is actually inserted into the slot. In this scenario, the caller may pass -1 for *slot\_nr*.

The following semantics are imposed when the caller passes *slot\_nr* == -1. First, we no longer check for an existing struct pci\_slot, as there may be many slots with *slot\_nr* of -1. The other change in semantics is user-visible, which is the 'address' parameter presented in sysfs will

## consist solely of a dddd

bb tuple, where dddd is the PCI domain of the struct pci\_bus and bb is the bus number. In other words, the devfn of the 'placeholder' slot will not be displayed.

## Name

`pci_destroy_slot` — decrement refcount for physical PCI slot

## Synopsis

```
void pci_destroy_slot (struct pci_slot * slot);
```

## Arguments

*slot*   struct pci\_slot to decrement

## Description

`struct pci_slot` is refcounted, so destroying them is really easy; we just call `kobject_put` on its `kobj` and let our release methods do the rest.

## Name

`pci_hp_create_module_link` — create symbolic link to the hotplug driver module.

## Synopsis

```
void pci_hp_create_module_link (struct pci_slot * pci_slot);
```

## Arguments

*pci\_slot*    struct pci\_slot

## Description

Helper function for `pci_hotplug_core.c` to create symbolic link to the hotplug driver module.

## Name

`pci_hp_remove_module_link` — remove symbolic link to the hotplug driver module.

## Synopsis

```
void pci_hp_remove_module_link (struct pci_slot * pci_slot);
```

## Arguments

*pci\_slot*   struct pci\_slot

## Description

Helper function for `pci_hotplug_core.c` to remove symbolic link to the hotplug driver module.



## Name

`pci_enable_rom` — enable ROM decoding for a PCI device

## Synopsis

```
int pci_enable_rom (struct pci_dev * pdev);
```

## Arguments

*pdev*    PCI device to enable

## Description

Enable ROM decoding on *dev*. This involves simply turning on the last bit of the PCI ROM BAR. Note that some cards may share address decoders between the ROM and other resources, so enabling it may disable access to MMIO registers or other card memory.

## Name

`pci_disable_rom` — disable ROM decoding for a PCI device

## Synopsis

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

## Arguments

*pdev*    PCI device to disable

## Description

Disable ROM decoding on a PCI device by turning off the last bit in the ROM BAR.

## Name

`pci_map_rom` — map a PCI ROM to kernel space

## Synopsis

```
void __iomem * pci_map_rom (struct pci_dev * pdev, size_t * size);
```

## Arguments

*pdev* pointer to pci device struct

*size* pointer to receive size of pci window over ROM

## Return

kernel virtual pointer to image of ROM

Map a PCI ROM into kernel space. If ROM is boot video ROM, the shadow BIOS copy will be returned instead of the actual ROM.

## Name

`pci_unmap_rom` — unmap the ROM from kernel space

## Synopsis

```
void pci_unmap_rom (struct pci_dev * pdev, void __iomem * rom);
```

## Arguments

*pdev*    pointer to pci device struct

*rom*     virtual address of the previous mapping

## Description

Remove a mapping of a previously mapped ROM

## Name

`pci_platform_rom` — provides a pointer to any ROM image provided by the platform

## Synopsis

```
void __iomem * pci_platform_rom (struct pci_dev * pdev, size_t * size);
```

## Arguments

*pdev* pointer to pci device struct

*size* pointer to receive size of pci window over ROM

## Name

`pci_enable_sriov` — enable the SR-IOV capability

## Synopsis

```
int pci_enable_sriov (struct pci_dev * dev, int nr_virtfn);
```

## Arguments

*dev*            the PCI device

*nr\_virtfn*    number of virtual functions to enable

## Description

Returns 0 on success, or negative on failure.

## Name

`pci_disable_sriov` — disable the SR-IOV capability

## Synopsis

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

## Arguments

*dev* the PCI device

## Name

`pci_num_vf` — return number of VFs associated with a PF device\_release\_driver

## Synopsis

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

## Arguments

*dev* the PCI device

## Description

Returns number of VFs, or 0 if SR-IOV is not enabled.



## Name

`pci_vfs_assigned` — returns number of VFs are assigned to a guest

## Synopsis

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

## Arguments

*dev* the PCI device

## Description

Returns number of VFs belonging to this device that are assigned to a guest. If device is not a physical function returns 0.

## Name

`pci_sriov_set_totalvfs` — - reduce the TotalVFs available

## Synopsis

```
int pci_sriov_set_totalvfs (struct pci_dev * dev, u16 numvfs);
```

## Arguments

*dev*        the PCI PF device

*numvfs*    number that should be used for TotalVFs supported

## Description

Should be called from PF driver's probe routine with device's mutex held.

Returns 0 if PF is an SRIOV-capable device and value of *numvfs* valid. If not a PF return -ENOSYS; if *numvfs* is invalid return -EINVAL; if VFs already enabled, return -EBUSY.

## Name

`pci_sriov_get_totalvfs` — - get total VFs supported on this device

## Synopsis

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

## Arguments

*dev* the PCI PF device

## Description

For a PCIe device with SRIOV support, return the PCIe SRIOV capability value of TotalVFs or the value of `driver_max_VFs` if the driver reduced it. Otherwise 0.

## Name

`pci_read_legacy_io` — read byte(s) from legacy I/O port space

## Synopsis

```
ssize_t pci_read_legacy_io (struct file * filp, struct kobject * kobj,  
struct bin_attribute * bin_attr, char * buf, loff_t off, size_t count);
```

## Arguments

<i>filp</i>	open sysfs file
<i>kobj</i>	kobject corresponding to file to read from
<i>bin_attr</i>	struct <code>bin_attribute</code> for this file
<i>buf</i>	buffer to store results
<i>off</i>	offset into legacy I/O port space
<i>count</i>	number of bytes to read

## Description

Reads 1, 2, or 4 bytes from legacy I/O port space using an arch specific callback routine (`pci_legacy_read`).

## Name

`pci_write_legacy_io` — write byte(s) to legacy I/O port space

## Synopsis

```
ssize_t pci_write_legacy_io (struct file * filp, struct kobject * kobj,  
struct bin_attribute * bin_attr, char * buf, loff_t off, size_t count);
```

## Arguments

<i>filp</i>	open sysfs file
<i>kobj</i>	kobject corresponding to file to read from
<i>bin_attr</i>	struct <code>bin_attribute</code> for this file
<i>buf</i>	buffer containing value to be written
<i>off</i>	offset into legacy I/O port space
<i>count</i>	number of bytes to write

## Description

Writes 1, 2, or 4 bytes from legacy I/O port space using an arch specific callback routine (`pci_legacy_write`).

## Name

`pci_mmap_legacy_mem` — map legacy PCI memory into user memory space

## Synopsis

```
int pci_mmap_legacy_mem (struct file * filp, struct kobject * kobj,
struct bin_attribute * attr, struct vm_area_struct * vma);
```

## Arguments

*filp* open sysfs file

*kobj* kobject corresponding to device to be mapped

*attr* struct bin\_attribute for this file

*vma* struct vm\_area\_struct passed to mmap

## Description

Uses an arch specific callback, `pci_mmap_legacy_mem_page_range`, to mmap legacy memory space (first meg of bus space) into application virtual memory space.

## Name

`pci_mmap_legacy_io` — map legacy PCI IO into user memory space

## Synopsis

```
int pci_mmap_legacy_io (struct file * filp, struct kobject * kobj,  
struct bin_attribute * attr, struct vm_area_struct * vma);
```

## Arguments

*filp* open sysfs file

*kobj* kobject corresponding to device to be mapped

*attr* struct bin\_attribute for this file

*vma* struct vm\_area\_struct passed to mmap

## Description

Uses an arch specific callback, `pci_mmap_legacy_io_page_range`, to mmap legacy IO space (first meg of bus space) into application virtual memory space. Returns `-ENOSYS` if the operation isn't supported

## Name

`pci_adjust_legacy_attr` — adjustment of legacy file attributes

## Synopsis

```
void pci_adjust_legacy_attr (struct pci_bus * b, enum pci_mmap_state  
mmap_type);
```

## Arguments

*b*                    bus to create files under

*mmap\_type*        I/O port or memory

## Description

Stub implementation. Can be overridden by arch if necessary.



## Name

`pci_create_legacy_files` — create legacy I/O port and memory files

## Synopsis

```
void pci_create_legacy_files (struct pci_bus * b);
```

## Arguments

*b* bus to create files under

## Description

Some platforms allow access to legacy I/O port and ISA memory space on a per-bus basis. This routine creates the files and ties them into their associated read, write and mmap files from `pci-sysfs.c`

On error unwind, but don't propagate the error to the caller as it is ok to set up the PCI bus without these files.

## Name

`pci_mmap_resource` — map a PCI resource into user memory space

## Synopsis

```
int pci_mmap_resource (struct kobject * kobj, struct bin_attribute *  
attr, struct vm_area_struct * vma, int write_combine);
```

## Arguments

<i>kobj</i>	kobject for mapping
<i>attr</i>	struct bin_attribute for the file being mapped
<i>vma</i>	struct vm_area_struct passed into the mmap
<i>write_combine</i>	1 for write_combine mapping

## Description

Use the regular PCI mapping routines to map a PCI resource into userspace.

## Name

`pci_remove_resource_files` — cleanup resource files

## Synopsis

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

## Arguments

*pdev*    dev to cleanup

## Description

If we created resource files for *pdev*, remove them from sysfs and free their resources.

## Name

`pci_create_resource_files` — create resource files in sysfs for *dev*

## Synopsis

```
int pci_create_resource_files (struct pci_dev * pdev);
```

## Arguments

*pdev*    dev in question

## Description

Walk the resources in *pdev* creating files for each resource available.

## Name

`pci_write_rom` — used to enable access to the PCI ROM display

## Synopsis

```
ssize_t pci_write_rom (struct file * filp, struct kobject * kobj, struct  
bin_attribute * bin_attr, char * buf, loff_t off, size_t count);
```

## Arguments

<i>filp</i>	sysfs file
<i>kobj</i>	kernel object handle
<i>bin_attr</i>	struct <code>bin_attribute</code> for this file
<i>buf</i>	user input
<i>off</i>	file offset
<i>count</i>	number of byte in input

## Description

writing anything except 0 enables it

## Name

`pci_read_rom` — read a PCI ROM

## Synopsis

```
ssize_t pci_read_rom (struct file * filp, struct kobject * kobj, struct  
bin_attribute * bin_attr, char * buf, loff_t off, size_t count);
```

## Arguments

<i>filp</i>	sysfs file
<i>kobj</i>	kernel object handle
<i>bin_attr</i>	struct <code>bin_attribute</code> for this file
<i>buf</i>	where to put the data we read from the ROM
<i>off</i>	file offset
<i>count</i>	number of bytes to read

## Description

Put *count* bytes starting at *off* into *buf* from the ROM in the PCI device corresponding to *kobj*.

## Name

`pci_remove_sysfs_dev_files` — cleanup PCI specific sysfs files

## Synopsis

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

## Arguments

*pdev* device whose entries we should free

## Description

Cleanup when *pdev* is removed from sysfs.

# PCI Hotplug Support Library

## Name

`__pci_hp_register` — register a `hotplug_slot` with the PCI hotplug subsystem

## Synopsis

```
int __pci_hp_register (struct hotplug_slot * slot, struct pci_bus *
bus, int devnr, const char * name, struct module * owner, const char
* mod_name);
```

## Arguments

<i>slot</i>	pointer to the struct <code>hotplug_slot</code> to register
<i>bus</i>	bus this slot is on
<i>devnr</i>	device number
<i>name</i>	name registered with kobject core
<i>owner</i>	caller module owner
<i>mod_name</i>	caller module name

## Description

Registers a hotplug slot with the pci hotplug subsystem, which will allow userspace interaction to the slot.

Returns 0 if successful, anything else for an error.



## Name

`pci_hp_deregister` — deregister a `hotplug_slot` with the PCI hotplug subsystem

## Synopsis

```
int pci_hp_deregister (struct hotplug_slot * slot);
```

## Arguments

*slot* pointer to the struct `hotplug_slot` to deregister

## Description

The *slot* must have been registered with the pci hotplug subsystem previously with a call to `pci_hp_register`.

Returns 0 if successful, anything else for an error.

## Name

`pci_hp_change_slot_info` — changes the slot's information structure in the core

## Synopsis

```
int pci_hp_change_slot_info (struct hotplug_slot * slot, struct
hotplug_slot_info * info);
```

## Arguments

*slot* pointer to the slot whose info has changed

*info* pointer to the info copy into the slot's info structure

## Description

*slot* must have been registered with the pci hotplug subsystem previously with a call to `pci_hp_register`.

Returns 0 if successful, anything else for an error.

---

# **Chapter 10. Firmware Interfaces**

## **DMI Interfaces**

## Name

`dmi_check_system` — check system DMI data

## Synopsis

```
int dmi_check_system (const struct dmi_system_id * list);
```

## Arguments

*list* array of `dmi_system_id` structures to match against All non-null elements of the list must match their slot's (field index's) data (i.e., each list string must be a substring of the specified DMI slot's string data) to be considered a successful match.

## Description

Walk the blacklist table running matching functions until someone returns non zero or we hit the end. Callback function is called for each successful match. Returns the number of matches.

## Name

`dmi_first_match` — find `dmi_system_id` structure matching system DMI data

## Synopsis

```
const struct dmi_system_id * dmi_first_match (const struct dmi_system_id
* list);
```

## Arguments

*list* array of `dmi_system_id` structures to match against All non-null elements of the list must match their slot's (field index's) data (i.e., each list string must be a substring of the specified DMI slot's string data) to be considered a successful match.

## Description

Walk the blacklist table until the first match is found. Return the pointer to the matching entry or NULL if there's no match.

## Name

`dmi_get_system_info` — return DMI data value

## Synopsis

```
const char * dmi_get_system_info (int field);
```

## Arguments

*field* data index (see enum `dmi_field`)

## Description

Returns one DMI data value, can be used to perform complex DMI data checks.

## Name

`dmi_name_in_vendors` — Check if string is in the DMI system or board vendor name

## Synopsis

```
int dmi_name_in_vendors (const char * str);
```

## Arguments

*str* Case sensitive Name

## Name

`dmi_find_device` — find onboard device by type/name

## Synopsis

```
const struct dmi_device * dmi_find_device (int type, const char * name,  
const struct dmi_device * from);
```

## Arguments

*type* device type or `DMI_DEV_TYPE_ANY` to match all device types

*name* device name string or `NULL` to match all

*from* previous device found in search, or `NULL` for new search.

## Description

Iterates through the list of known onboard devices. If a 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` as the *from* argument. If *from* is not `NULL`, searches continue from next device.



## Name

`dmi_get_date` — parse a DMI date

## Synopsis

```
bool dmi_get_date (int field, int * yearp, int * monthp, int * dayp);
```

## Arguments

*field*     data index (see enum `dmi_field`)

*yearp*    optional out parameter for the year

*monthp*   optional out parameter for the month

*dayp*     optional out parameter for the day

## Description

The date field is assumed to be in the form resembling `[mm[/dd]]/yy[yy]` and the result is stored in the out parameters any or all of which can be omitted.

If the field doesn't exist, all out parameters are set to zero and false is returned. Otherwise, true is returned with any invalid part of date set to zero.

On return, year, month and day are guaranteed to be in the range of `[0,9999]`, `[0,12]` and `[0,31]` respectively.

## Name

`dmi_walk` — Walk the DMI table and get called back for every record

## Synopsis

```
int dmi_walk (void (*decode) (const struct dmi_header *, void *), void  
* private_data);
```

## Arguments

*decode*                Callback function

*private\_data*    Private data to be passed to the callback function

## Description

Returns -1 when the DMI table can't be reached, 0 on success.

## Name

`dmi_match` — compare a string to the dmi field (if exists)

## Synopsis

```
bool dmi_match (enum dmi_field f, const char * str);
```

## Arguments

*f*      DMI field identifier

*str*    string to compare the DMI field to

## Description

Returns true if the requested field equals to the str (including NULL).

# EDD Interfaces

## Name

`edd_show_raw_data` — copies raw data to buffer for userspace to parse

## Synopsis

```
ssize_t edd_show_raw_data (struct edd_device * edev, char * buf);
```

## Arguments

*edev*    target `edd_device`

*buf*     output buffer

## Returns

number of bytes written, or `-EINVAL` on failure

## Name

edd\_release — free edd structure

## Synopsis

```
void edd_release (struct kobject * kobj);
```

## Arguments

*kobj* kobject of edd structure

## Description

This is called when the refcount of the edd structure reaches 0. This should happen right after we unregister, but just in case, we use the release callback anyway.

## Name

`edd_dev_is_type` — is this EDD device a 'type' device?

## Synopsis

```
int edd_dev_is_type (struct edd_device * edev, const char * type);
```

## Arguments

*edev*    target edd\_device

*type*    a host bus or interface identifier string per the EDD spec

## Description

Returns 1 (TRUE) if it is a 'type' device, 0 otherwise.

## Name

`edd_get_pci_dev` — finds `pci_dev` that matches `edev`

## Synopsis

```
struct pci_dev * edd_get_pci_dev (struct edd_device * edev);
```

## Arguments

*edev*   `edd_device`

## Description

Returns `pci_dev` if found, or `NULL`

## Name

`edd_init` — creates sysfs tree of EDD data

## Synopsis

```
int edd_init ( void );
```

## Arguments

*void* no arguments



---

# Chapter 11. Security Framework

## Name

`security_init` — initializes the security framework

## Synopsis

```
int security_init ( void );
```

## Arguments

*void* no arguments

## Description

This should be called early in the kernel initialization sequence.

## Name

`security_module_enable` — Load given security module on boot ?

## Synopsis

```
int security_module_enable (const char * module);
```

## Arguments

*module* the name of the module

## Description

Each LSM must pass this method before registering its own operations to avoid security registration races. This method may also be used to check if your LSM is currently loaded during kernel initialization.

## Return true if

-The passed LSM is the one chosen by user at boot time, -or the passed LSM is configured as the default and the user did not choose an alternate LSM at boot time. Otherwise, return false.

## Name

`securityfs_create_file` — create a file in the securityfs filesystem

## Synopsis

```
struct dentry * securityfs_create_file (const char * name, umode_t mode,
struct dentry * parent, void * data, const struct file_operations *
fops);
```

## Arguments

<i>name</i>	a pointer to a string containing the name of the file to create.
<i>mode</i>	the permission that the file should have
<i>parent</i>	a pointer to the parent dentry for this file. This should be a directory dentry if set. If this parameter is NULL, then the file will be created in the root of the securityfs filesystem.
<i>data</i>	a pointer to something that the caller will want to get to later on. The <code>inode.i_private</code> pointer will point to this value on the open call.
<i>fops</i>	a pointer to a struct <code>file_operations</code> that should be used for this file.

## Description

This is the basic “create a file” function for securityfs. It allows for a wide range of flexibility in creating a file, or a directory (if you want to create a directory, the `securityfs_create_dir` function is recommended to be used instead).

This function returns a pointer to a dentry if it succeeds. This pointer must be passed to the `securityfs_remove` function when the file is to be removed (no automatic cleanup happens if your module is unloaded, you are responsible here). If an error occurs, the function will return the error value (via `ERR_PTR`).

If securityfs is not enabled in the kernel, the value `-ENODEV` is returned.

## Name

`securityfs_create_dir` — create a directory in the securityfs filesystem

## Synopsis

```
struct dentry * securityfs_create_dir (const char * name, struct dentry  
* parent);
```

## Arguments

*name*      a pointer to a string containing the name of the directory to create.

*parent*    a pointer to the parent dentry for this file. This should be a directory dentry if set. If this parameter is `NULL`, then the directory will be created in the root of the securityfs filesystem.

## Description

This function creates a directory in securityfs with the given *name*.

This function returns a pointer to a dentry if it succeeds. This pointer must be passed to the `securityfs_remove` function when the file is to be removed (no automatic cleanup happens if your module is unloaded, you are responsible here). If an error occurs, `NULL` will be returned.

If securityfs is not enabled in the kernel, the value `-ENODEV` is returned. It is not wise to check for this value, but rather, check for `NULL` or `!NULL` instead as to eliminate the need for `#ifdef` in the calling code.

## Name

`securityfs_remove` — removes a file or directory from the securityfs filesystem

## Synopsis

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

## Arguments

*dentry* a pointer to a the dentry of the file or directory to be removed.

## Description

This function removes a file or directory in securityfs that was previously created with a call to another securityfs function (like `securityfs_create_file` or variants thereof.)

This function is required to be called in order for the file to be removed. No automatic cleanup of files will happen when a module is removed; you are responsible here.

---

## Chapter 12. Audit Interfaces

## Name

`audit_log_start` — obtain an audit buffer

## Synopsis

```
struct audit_buffer * audit_log_start (struct audit_context * ctx, gfp_t  
gfp_mask, int type);
```

## Arguments

*ctx*            audit\_context (may be NULL)

*gfp\_mask*    type of allocation

*type*        audit message type

## Description

Returns `audit_buffer` pointer on success or `NULL` on error.

Obtain an audit buffer. This routine does locking to obtain the audit buffer, but then no locking is required for calls to `audit_log_*format`. If the task (*ctx*) is a task that is currently in a syscall, then the syscall is marked as auditable and an audit record will be written at syscall exit. If there is no associated task, then task context (*ctx*) should be `NULL`.



## Name

`audit_log_format` — format a message into the audit buffer.

## Synopsis

```
void audit_log_format (struct audit_buffer * ab, const char * fmt, ...);
```

## Arguments

*ab*     audit\_buffer

*fmt*    format string @...: optional parameters matching *fmt* string

...     variable arguments

## Description

All the work is done in `audit_log_vformat`.

## Name

`audit_log_end` — end one audit record

## Synopsis

```
void audit_log_end (struct audit_buffer * ab);
```

## Arguments

*ab* the `audit_buffer`

## Description

`netlink_unicast` cannot be called inside an irq context because it blocks (last arg, `flags`, is not set to `MSG_DONTWAIT`), so the audit buffer is placed on a queue and a tasklet is scheduled to remove them from the queue outside the irq context. May be called in any context.

## Name

`audit_log` — Log an audit record

## Synopsis

```
void audit_log (struct audit_context * ctx, gfp_t gfp_mask, int type,  
const char * fmt, ...);
```

## Arguments

<i>ctx</i>	audit context
<i>gfp_mask</i>	type of allocation
<i>type</i>	audit message type
<i>fmt</i>	format string to use @...: variable parameters matching the format string
...	variable arguments

## Description

This is a convenience function that calls `audit_log_start`, `audit_log_vformat`, and `audit_log_end`. It may be called in any context.

## Name

`audit_log_secctx` — Converts and logs SELinux context

## Synopsis

```
void audit_log_secctx (struct audit_buffer * ab, u32 secid);
```

## Arguments

*ab*        audit\_buffer

*secid*    security number

## Description

This is a helper function that calls `security_secid_to_secctx` to convert `secid` to `secctx` and then adds the (converted) SELinux context to the audit log by calling `audit_log_format`, thus also preventing leak of internal `secid` to userspace. If `secid` cannot be converted `audit_panic` is called.

## Name

`audit_alloc` — allocate an audit context block for a task

## Synopsis

```
int audit_alloc (struct task_struct * tsk);
```

## Arguments

*tsk* task

## Description

Filter on the task information and allocate a per-task audit context if necessary. Doing so turns on system call auditing for the specified task. This is called from `copy_process`, so no lock is needed.

## Name

`__audit_free` — free a per-task audit context

## Synopsis

```
void __audit_free (struct task_struct * tsk);
```

## Arguments

*tsk* task whose audit context block to free

## Description

Called from `copy_process` and `do_exit`

## Name

`__audit_syscall_entry` — fill in an audit record at syscall entry

## Synopsis

```
void __audit_syscall_entry (int major, unsigned long a1, unsigned long  
a2, unsigned long a3, unsigned long a4);
```

## Arguments

*major*    major syscall type (function)

*a1*       additional syscall register 1

*a2*       additional syscall register 2

*a3*       additional syscall register 3

*a4*       additional syscall register 4

## Description

Fill in audit context at syscall entry. This only happens if the audit context was created when the task was created and the state or filters demand the audit context be built. If the state from the per-task filter or from the per-syscall filter is `AUDIT_RECORD_CONTEXT`, then the record will be written at syscall exit time (otherwise, it will only be written if another part of the kernel requests that it be written).

## Name

`__audit_syscall_exit` — deallocate audit context after a system call

## Synopsis

```
void __audit_syscall_exit (int success, long return_code);
```

## Arguments

*success*            success value of the syscall

*return\_code*    return value of the syscall

## Description

Tear down after system call. If the audit context has been marked as auditable (either because of the `AUDIT_RECORD_CONTEXT` state from filtering, or because some other part of the kernel wrote an audit message), then write out the syscall information. In call cases, free the names stored from `getname`.



## Name

`__audit_reuseusername` — fill out filename with info from existing entry

## Synopsis

```
struct filename * __audit_reuseusername (const __user char * uptr);
```

## Arguments

*uptr*    userland ptr to pathname

## Description

Search the `audit_names` list for the current audit context. If there is an existing entry with a matching “`uptr`” then return the filename associated with that `audit_name`. If not, return `NULL`.

## Name

`__audit_getname` — add a name to the list

## Synopsis

```
void __audit_getname (struct filename * name);
```

## Arguments

*name*   name to add

## Description

Add a name to the list of audit names for this context. Called from fs/namei.c:getname.

## Name

`__audit_inode` — store the inode and device from a lookup

## Synopsis

```
void __audit_inode (struct filename * name, const struct dentry * dentry,  
unsigned int flags);
```

## Arguments

*name*      name being audited

*dentry*    dentry being audited

*flags*      attributes for this particular entry

## Name

`auditsc_get_stamp` — get local copies of `audit_context` values

## Synopsis

```
int auditsc_get_stamp (struct audit_context * ctx, struct timespec *  
t, unsigned int * serial);
```

## Arguments

*ctx*        `audit_context` for the task

*t*         `timespec` to store time recorded in the `audit_context`

*serial*    serial value that is recorded in the `audit_context`

## Description

Also sets the context as auditable.

## Name

`audit_set_loginuid` — set current task's `audit_context` `loginuid`

## Synopsis

```
int audit_set_loginuid (kuid_t loginuid);
```

## Arguments

*loginuid* `loginuid` value

## Description

Returns 0.

Called (set) from `fs/proc/base.c::proc_loginuid_write`.

## Name

`__audit_mq_open` — record audit data for a POSIX MQ open

## Synopsis

```
void __audit_mq_open (int oflag, umode_t mode, struct mq_attr * attr);
```

## Arguments

*oflag*    open flag

*mode*    mode bits

*attr*    queue attributes

## Name

`__audit_mq_sendrecv` — record audit data for a POSIX MQ timed send/receive

## Synopsis

```
void __audit_mq_sendrecv (mqd_t mqdes, size_t msg_len, unsigned int  
msg_prio, const struct timespec * abs_timeout);
```

## Arguments

<i>mqdes</i>	MQ descriptor
<i>msg_len</i>	Message length
<i>msg_prio</i>	Message priority
<i>abs_timeout</i>	Message timeout in absolute time

## Name

`__audit_mq_notify` — record audit data for a POSIX MQ notify

## Synopsis

```
void __audit_mq_notify (mqd_t  mqdes,  const  struct  sigevent  *  
notification);
```

## Arguments

<i>mqdes</i>	MQ descriptor
<i>notification</i>	Notification event



## Name

`__audit_mq_getsetattr` — record audit data for a POSIX MQ get/set attribute

## Synopsis

```
void __audit_mq_getsetattr (mqd_t mqdes, struct mq_attr * mqstat);
```

## Arguments

*mqdes*    MQ descriptor

*mqstat*   MQ flags

## Name

`__audit_ipc_obj` — record audit data for ipc object

## Synopsis

```
void __audit_ipc_obj (struct kern_ipc_perm * ipcp);
```

## Arguments

*ipcp* ipc permissions

## Name

`__audit_ipc_set_perm` — record audit data for new ipc permissions

## Synopsis

```
void __audit_ipc_set_perm (unsigned long qbytes, uid_t uid, gid_t gid,  
umode_t mode);
```

## Arguments

*qbytes* msgq bytes

*uid* msgq user id

*gid* msgq group id

*mode* msgq mode (permissions)

## Description

Called only after `audit_ipc_obj`.

## Name

`__audit_socketcall` — record audit data for `sys_socketcall`

## Synopsis

```
int __audit_socketcall (int nargs, unsigned long * args);
```

## Arguments

*nargs*    number of args, which should not be more than `AUDITSC_ARGS`.

*args*    args array

## Name

`__audit_fd_pair` — record audit data for pipe and socketpair

## Synopsis

```
void __audit_fd_pair (int fd1, int fd2);
```

## Arguments

*fd1* the first file descriptor

*fd2* the second file descriptor

## Name

`__audit_sockaddr` — record audit data for `sys_bind`, `sys_connect`, `sys_sendto`

## Synopsis

```
int __audit_sockaddr (int len, void * a);
```

## Arguments

*len*    data length in user space

*a*      data address in kernel space

## Description

Returns 0 for success or NULL context or < 0 on error.

## Name

`__audit_signal_info` — record signal info for shutting down audit subsystem

## Synopsis

```
int __audit_signal_info (int sig, struct task_struct * t);
```

## Arguments

*sig*    signal value

*t*      task being signaled

## Description

If the audit subsystem is being terminated, record the task (pid) and uid that is doing that.

## Name

`__audit_log_bprm_fcaps` — store information about a loading bprm and relevant fcaps

## Synopsis

```
int __audit_log_bprm_fcaps (struct linux_binprm * bprm, const struct
cred * new, const struct cred * old);
```

## Arguments

*bprm* pointer to the bprm being processed

*new* the proposed new credentials

*old* the old credentials

## Description

Simply check if the proc already has the caps given by the file and if not store the priv escalation info for later auditing at the end of the syscall

-Eric



## Name

`__audit_log_capset` — store information about the arguments to the `capset` syscall

## Synopsis

```
void __audit_log_capset (const struct cred * new, const struct cred  
* old);
```

## Arguments

*new* the new credentials

*old* the old (current) credentials

## Description

Record the arguments userspace sent to `sys_capset` for later printing by the audit system if applicable

## Name

`audit_core_dumps` — record information about processes that end abnormally

## Synopsis

```
void audit_core_dumps (long signr);
```

## Arguments

*signr*    signal value

## Description

If a process ends with a core dump, something fishy is going on and we should record the event for investigation.

## Name

`audit_rule_change` — apply all rules to the specified message type

## Synopsis

```
int audit_rule_change (int type, __u32 portid, int seq, void * data,  
size_t datasz);
```

## Arguments

<i>type</i>	audit message type
<i>portid</i>	target port id for netlink audit messages
<i>seq</i>	netlink audit message sequence (serial) number
<i>data</i>	payload data
<i>datasz</i>	size of payload data

## Name

`audit_list_rules_send` — list the audit rules

## Synopsis

```
int audit_list_rules_send (struct sk_buff * request_skb, int seq);
```

## Arguments

*request\_skb*    skb of request we are replying to (used to target the reply)

*seq*            netlink audit message sequence (serial) number

## Name

`parent_len` — find the length of the parent portion of a pathname

## Synopsis

```
int parent_len (const char * path);
```

## Arguments

*path* pathname of which to determine length

## Name

`audit_compare_dname_path` — compare given dentry name with last component in given path. Return of 0 indicates a match.

## Synopsis

```
int audit_compare_dname_path (const char * dname, const char * path,  
int parentlen);
```

## Arguments

<i>dname</i>	dentry name that we're comparing
<i>path</i>	full pathname that we're comparing
<i>parentlen</i>	length of the parent if known. Passing in <code>AUDIT_NAME_FULL</code> here indicates that we must compute this value.

---

# Chapter 13. Accounting Framework

## Name

`sys_acct` — enable/disable process accounting

## Synopsis

```
long sys_acct (const char __user * name);
```

## Arguments

*name* file name for accounting records or NULL to shutdown accounting

## Description

Returns 0 for success or negative errno values for failure.

`sys_acct` is the only system call needed to implement process accounting. It takes the name of the file where accounting records should be written. If the filename is NULL, accounting will be shutdown.



## Name

`acct_collect` — collect accounting information into `pacct_struct`

## Synopsis

```
void acct_collect (long exitcode, int group_dead);
```

## Arguments

*exitcode*      task exit code

*group\_dead*   not 0, if this thread is the last one in the process.

## Name

`acct_process` —

## Synopsis

```
void acct_process ( void );
```

## Arguments

*void* no arguments

## Description

handles process accounting for an exiting task

---

# Chapter 14. Block Devices

## Name

`blk_get_backing_dev_info` — get the address of a queue's `backing_dev_info`

## Synopsis

```
struct backing_dev_info * blk_get_backing_dev_info (struct block_device  
* bdev);
```

## Arguments

*bdev* device

## Description

Locates the passed device's request queue and returns the address of its `backing_dev_info`. This function can only be called if *bdev* is opened and the return value is never NULL.

## Name

`blk_delay_queue` — restart queueing after defined interval

## Synopsis

```
void blk_delay_queue (struct request_queue * q, unsigned long msecs);
```

## Arguments

*q*            The struct `request_queue` in question

*msecs*      Delay in msec

## Description

Sometimes queueing needs to be postponed for a little while, to allow resources to come back. This function will make sure that queueing is restarted around the specified time. Queue lock must be held.

## Name

`blk_start_queue_async` — asynchronously restart a previously stopped queue

## Synopsis

```
void blk_start_queue_async (struct request_queue * q);
```

## Arguments

*q* The struct `request_queue` in question

## Description

`blk_start_queue_async` will clear the stop flag on the queue, and ensure that the `request_fn` for the queue is run from an async context.

## Name

`blk_start_queue` — restart a previously stopped queue

## Synopsis

```
void blk_start_queue (struct request_queue * q);
```

## Arguments

*q* The struct `request_queue` in question

## Description

`blk_start_queue` will clear the stop flag on the queue, and call the `request_fn` for the queue if it was in a stopped state when entered. Also see `blk_stop_queue`. Queue lock must be held.

## Name

`blk_stop_queue` — stop a queue

## Synopsis

```
void blk_stop_queue (struct request_queue * q);
```

## Arguments

*q* The struct request\_queue in question

## Description

The Linux block layer assumes that a block driver will consume all entries on the request queue when the request\_fn strategy is called. Often this will not happen, because of hardware limitations (queue depth settings). If a device driver gets a 'queue full' response, or if it simply chooses not to queue more I/O at one point, it can call this function to prevent the request\_fn from being called until the driver has signalled it's ready to go again. This happens by calling `blk_start_queue` to restart queue operations. Queue lock must be held.



## Name

`blk_sync_queue` — cancel any pending callbacks on a queue

## Synopsis

```
void blk_sync_queue (struct request_queue * q);
```

## Arguments

*q* the queue

## Description

The block layer may perform asynchronous callback activity on a queue, such as calling the `unplug` function after a timeout. A block device may call `blk_sync_queue` to ensure that any such activity is cancelled, thus allowing it to release resources that the callbacks might use. The caller must already have made sure that its `->make_request_fn` will not re-add plugging prior to calling this function.

This function does not cancel any asynchronous activity arising out of elevator or throttling code. That would require `elevator_exit` and `blkcg_exit_queue` to be called with queue lock initialized.

## Name

`__blk_run_queue_uncond` — run a queue whether or not it has been stopped

## Synopsis

```
void __blk_run_queue_uncond (struct request_queue * q);
```

## Arguments

*q* The queue to run

## Description

Invoke request handling on a queue if there are any pending requests. May be used to restart request handling after a request has completed. This variant runs the queue whether or not the queue has been stopped. Must be called with the queue lock held and interrupts disabled. See also *blk\_run\_queue*.

## Name

`__blk_run_queue` — run a single device queue

## Synopsis

```
void __blk_run_queue (struct request_queue * q);
```

## Arguments

*q*    The queue to run

## Description

See *blk\_run\_queue*. This variant must be called with the queue lock held and interrupts disabled.

## Name

`blk_run_queue_async` — run a single device queue in workqueue context

## Synopsis

```
void blk_run_queue_async (struct request_queue * q);
```

## Arguments

*q*    The queue to run

## Description

Tells kblockd to perform the equivalent of *blk\_run\_queue* on behalf of us. The caller must hold the queue lock.

## Name

`blk_run_queue` — run a single device queue

## Synopsis

```
void blk_run_queue (struct request_queue * q);
```

## Arguments

*q* The queue to run

## Description

Invoke request handling on this queue, if it has pending work to do. May be used to restart queueing when a request has completed.

## Name

`blk_queue_bypass_start` — enter queue bypass mode

## Synopsis

```
void blk_queue_bypass_start (struct request_queue * q);
```

## Arguments

*q* queue of interest

## Description

In bypass mode, only the dispatch FIFO queue of *q* is used. This function makes *q* enter bypass mode and drains all requests which were throttled or issued before. On return, it's guaranteed that no request is being throttled or has ELVPRIV set and `blk_queue_bypass` true inside queue or RCU read lock.

## Name

`blk_queue_bypass_end` — leave queue bypass mode

## Synopsis

```
void blk_queue_bypass_end (struct request_queue * q);
```

## Arguments

*q* queue of interest

## Description

Leave bypass mode and restore the normal queueing behavior.

## Name

`blk_cleanup_queue` — shutdown a request queue

## Synopsis

```
void blk_cleanup_queue (struct request_queue * q);
```

## Arguments

*q* request queue to shutdown

## Description

Mark *q* DYING, drain all pending requests, mark *q* DEAD, destroy and put it. All future requests will be failed immediately with -ENODEV.



## Name

`blk_init_queue` — prepare a request queue for use with a block device

## Synopsis

```
struct request_queue * blk_init_queue (request_fn_proc * rfn, spinlock_t  
* lock);
```

## Arguments

*rfn*     The function to be called to process requests that have been placed on the queue.

*lock*   Request queue spin lock

## Description

If a block device wishes to use the standard request handling procedures, which sorts requests and coalesces adjacent requests, then it must call `blk_init_queue`. The function *rfn* will be called when there are requests on the queue that need to be processed. If the device supports plugging, then *rfn* may not be called immediately when requests are available on the queue, but may be called at some time later instead. Plugged queues are generally unplugged when a buffer belonging to one of the requests on the queue is needed, or due to memory pressure.

*rfn* is not required, or even expected, to remove all requests off the queue, but only as many as it can handle at a time. If it does leave requests on the queue, it is responsible for arranging that the requests get dealt with eventually.

The queue spin lock must be held while manipulating the requests on the request queue; this lock will be taken also from interrupt context, so irq disabling is needed for it.

Function returns a pointer to the initialized request queue, or NULL if it didn't succeed.

## Note

`blk_init_queue` must be paired with a `blk_cleanup_queue` call when the block device is deactivated (such as at module unload).

## Name

`blk_make_request` — given a bio, allocate a corresponding struct request.

## Synopsis

```
struct request * blk_make_request (struct request_queue * q, struct bio
* bio, gfp_t gfp_mask);
```

## Arguments

*q*            target request queue

*bio*            The bio describing the memory mappings that will be submitted for IO. It may be a chained-bio properly constructed by block/bio layer.

*gfp\_mask*    gfp flags to be used for memory allocation

## Description

`blk_make_request` is the parallel of `generic_make_request` for `BLOCK_PC` type commands. Where the struct request needs to be farther initialized by the caller. It is passed a struct bio, which describes the memory info of the I/O transfer.

The caller of `blk_make_request` must make sure that `bi_io_vec` are set to describe the memory buffers. That `bio_data_dir` will return the needed direction of the request. (And all bio's in the passed bio-chain are properly set accordingly)

If called under none-sleepable conditions, mapped bio buffers must not need bouncing, by calling the appropriate masked or flagged allocator, suitable for the target device. Otherwise the call to `blk_queue_bounce` will BUG.

## WARNING

When allocating/cloning a bio-chain, careful consideration should be given to how you allocate bios. In particular, you cannot use `__GFP_DIRECT_RECLAIM` for anything but the first bio in the chain. Otherwise you risk waiting for IO completion of a bio that hasn't been submitted yet, thus resulting in a deadlock. Alternatively bios should be allocated using `bio_kmalloc` instead of `bio_alloc`, as that avoids the mempool deadlock. If possible a big IO should be split into smaller parts when allocation fails. Partial allocation should not be an error, or you risk a live-lock.

## Name

`blk_rq_set_block_pc` — initialize a request to type `BLOCK_PC`

## Synopsis

```
void blk_rq_set_block_pc (struct request * rq);
```

## Arguments

*rq* request to be initialized

## Name

`blk_requeue_request` — put a request back on queue

## Synopsis

```
void blk_requeue_request (struct request_queue * q, struct request *  
rq);
```

## Arguments

*q* request queue where request should be inserted

*rq* request to be inserted

## Description

Drivers often keep queueing requests until the hardware cannot accept more, when that condition happens we need to put the request back on the queue. Must be called with queue lock held.

## Name

`part_round_stats` — Round off the performance stats on a struct `disk_stats`.

## Synopsis

```
void part_round_stats (int cpu, struct hd_struct * part);
```

## Arguments

*cpu*     cpu number for stats access

*part*    target partition

## Description

The average IO queue length and utilisation statistics are maintained by observing the current state of the queue length and the amount of time it has been in this state for.

Normally, that accounting is done on IO completion, but that can result in more than a second's worth of IO being accounted for within any one second, leading to >100% utilisation. To deal with that, we call this function to do a round-off before returning the results when reading `/proc/diskstats`. This accounts immediately for all queue usage up to the current jiffies and restarts the counters again.

## Name

`blk_add_request_payload` — add a payload to a request

## Synopsis

```
void blk_add_request_payload (struct request * rq, struct page * page,  
unsigned int len);
```

## Arguments

*rq*      request to update

*page*    page backing the payload

*len*     length of the payload.

## Description

This allows to later add a payload to an already submitted request by a block driver. The driver needs to take care of freeing the payload itself.

Note that this is a quite horrible hack and nothing but handling of discard requests should ever use it.

## Name

`generic_make_request` — hand a buffer to its device driver for I/O

## Synopsis

```
blk_qc_t generic_make_request (struct bio * bio);
```

## Arguments

*bio* The bio describing the location in memory and on the device.

## Description

`generic_make_request` is used to make I/O requests of block devices. It is passed a struct bio, which describes the I/O that needs to be done.

`generic_make_request` does not return any status. The success/failure status of the request, along with notification of completion, is delivered asynchronously through the `bio->bi_end_io` function described (one day) else where.

The caller of `generic_make_request` must make sure that `bi_io_vec` are set to describe the memory buffer, and that `bi_dev` and `bi_sector` are set to describe the device address, and the `bi_end_io` and optionally `bi_private` are set to describe how completion notification should be signaled.

`generic_make_request` and the drivers it calls may use `bi_next` if this bio happens to be merged with someone else, and may resubmit the bio to a lower device by calling into `generic_make_request` recursively, which means the bio should NOT be touched after the call to `->make_request_fn`.

## Name

`submit_bio` — submit a bio to the block device layer for I/O

## Synopsis

```
blk_qc_t submit_bio (int rw, struct bio * bio);
```

## Arguments

*rw*     whether to READ or WRITE, or maybe to READA (read ahead)

*bio*    The struct bio which describes the I/O

## Description

`submit_bio` is very similar in purpose to `generic_make_request`, and uses that function to do most of the work. Both are fairly rough interfaces; *bio* must be presetup and ready for I/O.



## Name

`blk_insert_cloned_request` — Helper for stacking drivers to submit a request

## Synopsis

```
int blk_insert_cloned_request (struct request_queue * q, struct request  
* rq);
```

## Arguments

*q*    the queue to submit the request

*rq*   the request being queued

## Name

`blk_rq_err_bytes` — determine number of bytes till the next failure boundary

## Synopsis

```
unsigned int blk_rq_err_bytes (const struct request * rq);
```

## Arguments

*rq* request to examine

## Description

A request could be merge of IOs which require different failure handling. This function determines the number of bytes which can be failed from the beginning of the request without crossing into area which need to be retried further.

## Return

The number of bytes to fail.

## Context

`queue_lock` must be held.

## Name

`blk_peek_request` — peek at the top of a request queue

## Synopsis

```
struct request * blk_peek_request (struct request_queue * q);
```

## Arguments

*q* request queue to peek at

## Description

Return the request at the top of *q*. The returned request should be started using `blk_start_request` before LLD starts processing it.

## Return

Pointer to the request at the top of *q* if available. Null otherwise.

## Context

`queue_lock` must be held.

## Name

`blk_start_request` — start request processing on the driver

## Synopsis

```
void blk_start_request (struct request * req);
```

## Arguments

*req* request to dequeue

## Description

Dequeue *req* and start timeout timer on it. This hands off the request to the driver.

Block internal functions which don't want to start timer should call `blk_dequeue_request`.

## Context

`queue_lock` must be held.

## Name

`blk_fetch_request` — fetch a request from a request queue

## Synopsis

```
struct request * blk_fetch_request (struct request_queue * q);
```

## Arguments

*q* request queue to fetch a request from

## Description

Return the request at the top of *q*. The request is started on return and LLD can start processing it immediately.

## Return

Pointer to the request at the top of *q* if available. Null otherwise.

## Context

`queue_lock` must be held.

## Name

`blk_update_request` — Special helper function for request stacking drivers

## Synopsis

```
bool blk_update_request (struct request * req, int error, unsigned int
nr_bytes);
```

## Arguments

*req*            the request being processed

*error*        0 for success, < 0 for error

*nr\_bytes*    number of bytes to complete *req*

## Description

Ends I/O on a number of bytes attached to *req*, but doesn't complete the request structure even if *req* doesn't have leftover. If *req* has leftover, sets it up for the next range of segments.

This special helper function is only for request stacking drivers (e.g. request-based dm) so that they can handle partial completion. Actual device drivers should use `blk_end_request` instead.

Passing the result of `blk_rq_bytes` as *nr\_bytes* guarantees `false` return from this function.

## Return

`false` - this request doesn't have any more data `true` - this request has more data

## Name

`blk_unprep_request` — unprepare a request

## Synopsis

```
void blk_unprep_request (struct request * req);
```

## Arguments

*req* the request

## Description

This function makes a request ready for complete resubmission (or completion). It happens only after all error handling is complete, so represents the appropriate moment to deallocate any resources that were allocated to the request in the `prep_rq_fn`. The queue lock is held when calling this.

## Name

`blk_end_request` — Helper function for drivers to complete the request.

## Synopsis

```
bool blk_end_request (struct request * rq, int error, unsigned int  
nr_bytes);
```

## Arguments

*rq*            the request being processed

*error*        0 for success, < 0 for error

*nr\_bytes*    number of bytes to complete

## Description

Ends I/O on a number of bytes attached to *rq*. If *rq* has leftover, sets it up for the next range of segments.

## Return

`false` - we are done with this request `true` - still buffers pending for this request



## Name

`blk_end_request_all` — Helper function for drives to finish the request.

## Synopsis

```
void blk_end_request_all (struct request * rq, int error);
```

## Arguments

*rq*        the request to finish

*error*    0 for success, < 0 for error

## Description

Completely finish *rq*.

## Name

`blk_end_request_cur` — Helper function to finish the current request chunk.

## Synopsis

```
bool blk_end_request_cur (struct request * rq, int error);
```

## Arguments

*rq*        the request to finish the current chunk for

*error*    0 for success, < 0 for error

## Description

Complete the current consecutively mapped chunk from *rq*.

## Return

`false` - we are done with this request `true` - still buffers pending for this request

## Name

`blk_end_request_err` — Finish a request till the next failure boundary.

## Synopsis

```
bool blk_end_request_err (struct request * rq, int error);
```

## Arguments

*rq*        the request to finish till the next failure boundary for

*error*    must be negative errno

## Description

Complete *rq* till the next failure boundary.

## Return

false - we are done with this request true - still buffers pending for this request

## Name

`__blk_end_request` — Helper function for drivers to complete the request.

## Synopsis

```
bool __blk_end_request (struct request * rq, int error, unsigned int  
nr_bytes);
```

## Arguments

*rq*            the request being processed

*error*        0 for success, < 0 for error

*nr\_bytes*    number of bytes to complete

## Description

Must be called with queue lock held unlike `blk_end_request`.

## Return

`false` - we are done with this request `true` - still buffers pending for this request

## Name

`__blk_end_request_all` — Helper function for drives to finish the request.

## Synopsis

```
void __blk_end_request_all (struct request * rq, int error);
```

## Arguments

*rq*        the request to finish

*error*    0 for success, < 0 for error

## Description

Completely finish *rq*. Must be called with queue lock held.

## Name

`__blk_end_request_cur` — Helper function to finish the current request chunk.

## Synopsis

```
bool __blk_end_request_cur (struct request * rq, int error);
```

## Arguments

*rq*        the request to finish the current chunk for

*error*    0 for success, < 0 for error

## Description

Complete the current consecutively mapped chunk from *rq*. Must be called with queue lock held.

## Return

false - we are done with this request true - still buffers pending for this request

## Name

`__blk_end_request_err` — Finish a request till the next failure boundary.

## Synopsis

```
bool __blk_end_request_err (struct request * rq, int error);
```

## Arguments

*rq*        the request to finish till the next failure boundary for

*error*    must be negative errno

## Description

Complete *rq* till the next failure boundary. Must be called with queue lock held.

## Return

false - we are done with this request true - still buffers pending for this request

## Name

`rq_flush_dcache_pages` — Helper function to flush all pages in a request

## Synopsis

```
void rq_flush_dcache_pages (struct request * rq);
```

## Arguments

*rq* the request to be flushed

## Description

Flush all pages in *rq*.



## Name

`blk_lld_busy` — Check if underlying low-level drivers of a device are busy

## Synopsis

```
int blk_lld_busy (struct request_queue * q);
```

## Arguments

`q` the queue of the device being checked

## Description

Check if underlying low-level drivers of a device are busy. If the drivers want to export their busy state, they must set own exporting function using `blk_queue_lld_busy` first.

Basically, this function is used only by request stacking drivers to stop dispatching requests to underlying devices when underlying devices are busy. This behavior helps more I/O merging on the queue of the request stacking driver and prevents I/O throughput regression on burst I/O load.

## Return

0 - Not busy (The request stacking driver should dispatch request) 1 - Busy (The request stacking driver should stop dispatching request)

## Name

`blk_rq_unprep_clone` — Helper function to free all bios in a cloned request

## Synopsis

```
void blk_rq_unprep_clone (struct request * rq);
```

## Arguments

*rq* the clone request to be cleaned up

## Description

Free all bios in *rq* for a cloned request.

## Name

`blk_rq_prep_clone` — Helper function to setup clone request

## Synopsis

```
int blk_rq_prep_clone (struct request * rq, struct request * rq_src,
struct bio_set * bs, gfp_t gfp_mask, int (*bio_ctr) (struct bio *,
struct bio *, void *), void * data);
```

## Arguments

<i>rq</i>	the request to be setup
<i>rq_src</i>	original request to be cloned
<i>bs</i>	bio_set that bios for clone are allocated from
<i>gfp_mask</i>	memory allocation mask for bio
<i>bio_ctr</i>	setup function to be called for each clone bio. Returns 0 for success, non 0 for failure.
<i>data</i>	private data to be passed to <i>bio_ctr</i>

## Description

Clones bios in *rq\_src* to *rq*, and copies attributes of *rq\_src* to *rq*. The actual data parts of *rq\_src* (e.g. *->cmd*, *->sense*) are not copied, and copying such parts is the caller's responsibility. Also, pages which the original bios are pointing to are not copied and the cloned bios just point same pages. So cloned bios must be completed before original bios, which means the caller must complete *rq* before *rq\_src*.

## Name

`blk_start_plug` — initialize `blk_plug` and track it inside the `task_struct`

## Synopsis

```
void blk_start_plug (struct blk_plug * plug);
```

## Arguments

*plug* The struct `blk_plug` that needs to be initialized

## Description

Tracking `blk_plug` inside the `task_struct` will help with auto-flushing the pending I/O should the task end up blocking between `blk_start_plug` and `blk_finish_plug`. This is important from a performance perspective, but also ensures that we don't deadlock. For instance, if the task is blocking for a memory allocation, memory reclaim could end up wanting to free a page belonging to that request that is currently residing in our private plug. By flushing the pending I/O when the process goes to sleep, we avoid this kind of deadlock.

## Name

`blk_pm_runtime_init` — Block layer runtime PM initialization routine

## Synopsis

```
void blk_pm_runtime_init (struct request_queue * q, struct device *  
dev);
```

## Arguments

*q*      the queue of the device

*dev*    the device the queue belongs to

## Description

Initialize runtime-PM-related fields for *q* and start auto suspend for *dev*. Drivers that want to take advantage of request-based runtime PM should call this function after *dev* has been initialized, and its request queue *q* has been allocated, and runtime PM for it can not happen yet(either due to disabled/forbidden or its `usage_count > 0`). In most cases, driver should call this function before any I/O has taken place.

This function takes care of setting up using auto suspend for the device, the autosuspend delay is set to -1 to make runtime suspend impossible until an updated value is either set by user or by driver. Drivers do not need to touch other autosuspend settings.

The block layer runtime PM is request based, so only works for drivers that use request as their IO unit instead of those directly use bio's.

## Name

`blk_pre_runtime_suspend` — Pre runtime suspend check

## Synopsis

```
int blk_pre_runtime_suspend (struct request_queue * q);
```

## Arguments

*q* the queue of the device

## Description

This function will check if runtime suspend is allowed for the device by examining if there are any requests pending in the queue. If there are requests pending, the device can not be runtime suspended; otherwise, the queue's status will be updated to SUSPENDING and the driver can proceed to suspend the device.

For the not allowed case, we mark last busy for the device so that runtime PM core will try to autosuspend it some time later.

This function should be called near the start of the device's runtime\_suspend callback.

## Return

0 - OK to runtime suspend the device -EBUSY - Device should not be runtime suspended

## Name

`blk_post_runtime_suspend` — Post runtime suspend processing

## Synopsis

```
void blk_post_runtime_suspend (struct request_queue * q, int err);
```

## Arguments

*q*      the queue of the device

*err*    return value of the device's `runtime_suspend` function

## Description

Update the queue's runtime status according to the return value of the device's `runtime_suspend` function and mark last busy for the device so that PM core will try to auto suspend the device at a later time.

This function should be called near the end of the device's `runtime_suspend` callback.

## Name

`blk_pre_runtime_resume` — Pre runtime resume processing

## Synopsis

```
void blk_pre_runtime_resume (struct request_queue * q);
```

## Arguments

*q* the queue of the device

## Description

Update the queue's runtime status to RESUMING in preparation for the runtime resume of the device.

This function should be called near the start of the device's `runtime_resume` callback.



## Name

`blk_post_runtime_resume` — Post runtime resume processing

## Synopsis

```
void blk_post_runtime_resume (struct request_queue * q, int err);
```

## Arguments

*q*      the queue of the device

*err*    return value of the device's `runtime_resume` function

## Description

Update the queue's runtime status according to the return value of the device's `runtime_resume` function. If it is successfully resumed, process the requests that are queued into the device's queue when it is resuming and then mark last busy and initiate autosuspend for it.

This function should be called near the end of the device's `runtime_resume` callback.

## Name

`__blk_drain_queue` — drain requests from request\_queue

## Synopsis

```
void __blk_drain_queue (struct request_queue * q, bool drain_all);
```

## Arguments

*q*                    queue to drain

*drain\_all*    whether to drain all requests or only the ones w/ ELVPRIV

## Description

Drain requests from *q*. If *drain\_all* is set, all requests are drained. If not, only ELVPRIV requests are drained. The caller is responsible for ensuring that no new requests which need to be drained are queued.

## Name

`rq_ioc` — determine `io_context` for request allocation

## Synopsis

```
struct io_context * rq_ioc (struct bio * bio);
```

## Arguments

*bio* request being allocated is for this bio (can be NULL)

## Description

Determine `io_context` to use for request allocation for *bio*. May return NULL if `current->io_context` doesn't exist.

## Name

`__get_request` — get a free request

## Synopsis

```
struct request * __get_request (struct request_list * rl, int rw_flags,  
struct bio * bio, gfp_t gfp_mask);
```

## Arguments

<i>rl</i>	request list to allocate from
<i>rw_flags</i>	RW and SYNC flags
<i>bio</i>	bio to allocate request for (can be NULL)
<i>gfp_mask</i>	allocation mask

## Description

Get a free request from *q*. This function may fail under memory pressure or if *q* is dead.

Must be called with *q*->queue\_lock held and, Returns ERR\_PTR on failure, with *q*->queue\_lock held. Returns request pointer on success, with *q*->queue\_lock \*not held\*.

## Name

`get_request` — get a free request

## Synopsis

```
struct request * get_request (struct request_queue * q, int rw_flags,  
struct bio * bio, gfp_t gfp_mask);
```

## Arguments

<i>q</i>	request_queue to allocate request from
<i>rw_flags</i>	RW and SYNC flags
<i>bio</i>	bio to allocate request for (can be NULL)
<i>gfp_mask</i>	allocation mask

## Description

Get a free request from *q*. If `__GFP_DIRECT_RECLAIM` is set in *gfp\_mask*, this function keeps retrying under memory pressure and fails iff *q* is dead.

Must be called with *q*->queue\_lock held and, Returns `ERR_PTR` on failure, with *q*->queue\_lock held. Returns request pointer on success, with *q*->queue\_lock \*not held\*.

## Name

`blk_attempt_plug_merge` — try to merge with current's plugged list

## Synopsis

```
bool blk_attempt_plug_merge (struct request_queue * q, struct bio * bio,
unsigned int * request_count, struct request ** same_queue_rq);
```

## Arguments

<i>q</i>	request_queue new bio is being queued at
<i>bio</i>	new bio being queued
<i>request_count</i>	out parameter for number of traversed plugged requests
<i>same_queue_rq</i>	pointer to struct request that gets filled in when another request associated with <i>q</i> is found on the plug list (optional, may be NULL)

## Description

Determine whether *bio* being queued on *q* can be merged with a request on current's plugged list. Returns `true` if merge was successful, otherwise `false`.

Plugging coalesces IOs from the same issuer for the same purpose without going through *q->queue\_lock*. As such it's more of an issuing mechanism than scheduling, and the request, while may have `elvpriv` data, is not added on the elevator at this point. In addition, we don't have reliable access to the elevator outside queue lock. Only check basic merging parameters without querying the elevator.

Caller must ensure `!blk_queue_nomerges(q)` beforehand.

## Name

`blk_cloned_rq_check_limits` — Helper function to check a cloned request for new the queue limits

## Synopsis

```
int blk_cloned_rq_check_limits (struct request_queue * q, struct request
* rq);
```

## Arguments

*q* the queue

*rq* the request being checked

## Description

*rq* may have been made based on weaker limitations of upper-level queues in request stacking drivers, and it may violate the limitation of *q*. Since the block layer and the underlying device driver trust *rq* after it is inserted to *q*, it should be checked against *q* before the insertion using this generic function.

Request stacking drivers like request-based dm may change the queue limits when retrying requests on other queues. Those requests need to be checked against the new queue limits again during dispatch.

## Name

`blk_end_bidi_request` — Complete a bidi request

## Synopsis

```
bool blk_end_bidi_request (struct request * rq, int error, unsigned int  
nr_bytes, unsigned int bidi_bytes);
```

## Arguments

<i>rq</i>	the request to complete
<i>error</i>	0 for success, < 0 for error
<i>nr_bytes</i>	number of bytes to complete <i>rq</i>
<i>bidi_bytes</i>	number of bytes to complete <i>rq</i> ->next_rq

## Description

Ends I/O on a number of bytes attached to *rq* and *rq*->next\_rq. Drivers that supports bidi can safely call this member for any type of request, bidi or uni. In the later case *bidi\_bytes* is just ignored.

## Return

`false` - we are done with this request `true` - still buffers pending for this request



## Name

`__blk_end_bidi_request` — Complete a bidi request with queue lock held

## Synopsis

```
bool __blk_end_bidi_request (struct request * rq, int error, unsigned
int nr_bytes, unsigned int bidi_bytes);
```

## Arguments

<i>rq</i>	the request to complete
<i>error</i>	0 for success, < 0 for error
<i>nr_bytes</i>	number of bytes to complete <i>rq</i>
<i>bidi_bytes</i>	number of bytes to complete <i>rq</i> ->next_rq

## Description

Identical to `blk_end_bidi_request` except that queue lock is assumed to be locked on entry and remains so on return.

## Return

`false` - we are done with this request `true` - still buffers pending for this request

## Name

`blk_rq_map_user_iov` — map user data to a request, for `REQ_TYPE_BLOCK_PC` usage

## Synopsis

```
int blk_rq_map_user_iov (struct request_queue * q, struct request *
rq, struct rq_map_data * map_data, const struct iovec * iter, gfp_t
gfp_mask);
```

## Arguments

<i>q</i>	request queue where request should be inserted
<i>rq</i>	request to map data to
<i>map_data</i>	pointer to the <code>rq_map_data</code> holding pages (if necessary)
<i>iter</i>	iovec iterator
<i>gfp_mask</i>	memory allocation flags

## Description

Data will be mapped directly for zero copy I/O, if possible. Otherwise a kernel bounce buffer is used.

A matching `blk_rq_unmap_user` must be issued at the end of I/O, while still in process context.

## Note

The mapped bio may need to be bounced through `blk_queue_bounce` before being submitted to the device, as pages mapped may be out of reach. It's the callers responsibility to make sure this happens. The original bio must be passed back in to `blk_rq_unmap_user` for proper unmapping.

## Name

`blk_rq_unmap_user` — unmap a request with user data

## Synopsis

```
int blk_rq_unmap_user (struct bio * bio);
```

## Arguments

*bio*    start of bio list

## Description

Unmap a `rq` previously mapped by `blk_rq_map_user`. The caller must supply the original `rq->bio` from the `blk_rq_map_user` return, since the I/O completion may have changed `rq->bio`.

## Name

`blk_rq_map_kern` — map kernel data to a request, for `REQ_TYPE_BLOCK_PC` usage

## Synopsis

```
int blk_rq_map_kern (struct request_queue * q, struct request * rq, void
* kbuf, unsigned int len, gfp_t gfp_mask);
```

## Arguments

*q*                request queue where request should be inserted

*rq*               request to fill

*kbuf*            the kernel buffer

*len*             length of user data

*gfp\_mask*       memory allocation flags

## Description

Data will be mapped directly if possible. Otherwise a bounce buffer is used. Can be called multiple times to append multiple buffers.

## Name

`blk_release_queue` — release a struct `request_queue` when it is no longer needed

## Synopsis

```
void blk_release_queue (struct kobject * kobj);
```

## Arguments

*kobj* the `kobj` belonging to the request queue to be released

## Description

`blk_release_queue` is the pair to `blk_init_queue` or `blk_queue_make_request`. It should be called when a request queue is being released; typically when a block device is being de-registered. Currently, its primary task is to free all the struct request structures that were allocated to the queue and the queue itself.

## Note

The low level driver must have finished any outstanding requests first via `blk_cleanup_queue`.

## Name

`blk_queue_prep_rq` — set a `prepare_request` function for queue

## Synopsis

```
void blk_queue_prep_rq (struct request_queue * q, prep_rq_fn * pfn);
```

## Arguments

*q*      queue

*pfn*    `prepare_request` function

## Description

It's possible for a queue to register a `prepare_request` callback which is invoked before the request is handed to the `request_fn`. The goal of the function is to prepare a request for I/O, it can be used to build a cdb from the request data for instance.

## Name

`blk_queue_unprep_rq` — set an `unprepare_request` function for queue

## Synopsis

```
void blk_queue_unprep_rq (struct request_queue * q, unprep_rq_fn * ufn);
```

## Arguments

*q*      queue

*ufn*    `unprepare_request` function

## Description

It's possible for a queue to register an `unprepare_request` callback which is invoked before the request is finally completed. The goal of the function is to deallocate any data that was allocated in the `prepare_request` callback.

## Name

`blk_set_default_limits` — reset limits to default values

## Synopsis

```
void blk_set_default_limits (struct queue_limits * lim);
```

## Arguments

*lim* the queue\_limits structure to reset

## Description

Returns a queue\_limit struct to its default state.



## Name

`blk_set_stacking_limits` — set default limits for stacking devices

## Synopsis

```
void blk_set_stacking_limits (struct queue_limits * lim);
```

## Arguments

*lim* the queue\_limits structure to reset

## Description

Returns a queue\_limit struct to its default state. Should be used by stacking drivers like DM that have no internal limits.

## Name

`blk_queue_make_request` — define an alternate `make_request` function for a device

## Synopsis

```
void blk_queue_make_request (struct request_queue * q, make_request_fn  
* mfn);
```

## Arguments

*q*      the request queue for the device to be affected

*mfn*    the alternate `make_request` function

## Description

The normal way for struct bios to be passed to a device driver is for them to be collected into requests on a request queue, and then to allow the device driver to select requests off that queue when it is ready. This works well for many block devices. However some block devices (typically virtual devices such as md or lvm) do not benefit from the processing on the request queue, and are served best by having the requests passed directly to them. This can be achieved by providing a function to `blk_queue_make_request`.

## Caveat

The driver that does this *must* be able to deal appropriately with buffers in “highmemory”. This can be accomplished by either calling `__bio_kmap_atomic` to get a temporary kernel mapping, or by calling `blk_queue_bounce` to create a buffer in normal memory.

## Name

`blk_queue_bounce_limit` — set bounce buffer limit for queue

## Synopsis

```
void blk_queue_bounce_limit (struct request_queue * q, u64 max_addr);
```

## Arguments

*q*                    the request queue for the device

*max\_addr*           the maximum address the device can handle

## Description

Different hardware can have different requirements as to what pages it can do I/O directly to. A low level driver can call `blk_queue_bounce_limit` to have lower memory pages allocated as bounce buffers for doing I/O to pages residing above *max\_addr*.

## Name

`blk_queue_max_hw_sectors` — set max sectors for a request for this queue

## Synopsis

```
void blk_queue_max_hw_sectors (struct request_queue * q, unsigned int
max_hw_sectors);
```

## Arguments

*q*                      the request queue for the device

*max\_hw\_sectors*      max hardware sectors in the usual 512b unit

## Description

Enables a low level driver to set a hard upper limit, `max_hw_sectors`, on the size of requests. `max_hw_sectors` is set by the device driver based upon the capabilities of the I/O controller.

`max_dev_sectors` is a hard limit imposed by the storage device for READ/WRITE requests. It is set by the disk driver.

`max_sectors` is a soft limit imposed by the block layer for filesystem type requests. This value can be overridden on a per-device basis in `/sys/block/<device>/queue/max_sectors_kb`. The soft limit can not exceed `max_hw_sectors`.

## Name

`blk_queue_chunk_sectors` — set size of the chunk for this queue

## Synopsis

```
void blk_queue_chunk_sectors (struct request_queue * q, unsigned int
chunk_sectors);
```

## Arguments

*q*                      the request queue for the device

*chunk\_sectors*      chunk sectors in the usual 512b unit

## Description

If a driver doesn't want IOs to cross a given chunk size, it can set this limit and prevent merging across chunks. Note that the chunk size must currently be a power-of-2 in sectors. Also note that the block layer must accept a page worth of data at any offset. So if the crossing of chunks is a hard limitation in the driver, it must still be prepared to split single page bios.

## Name

`blk_queue_max_discard_sectors` — set max sectors for a single discard

## Synopsis

```
void blk_queue_max_discard_sectors (struct request_queue * q, unsigned  
int max_discard_sectors);
```

## Arguments

*q*                                      the request queue for the device

*max\_discard\_sectors*    maximum number of sectors to discard

## Name

`blk_queue_max_write_same_sectors` — set max sectors for a single write same

## Synopsis

```
void blk_queue_max_write_same_sectors (struct request_queue * q,  
unsigned int max_write_same_sectors);
```

## Arguments

*q* the request queue for the device

*max\_write\_same\_sectors* maximum number of sectors to write per command

## Name

`blk_queue_max_segments` — set max hw segments for a request for this queue

## Synopsis

```
void blk_queue_max_segments (struct request_queue * q, unsigned short  
max_segments);
```

## Arguments

*q*                      the request queue for the device

*max\_segments*    max number of segments

## Description

Enables a low level driver to set an upper limit on the number of hw data segments in a request.



## Name

`blk_queue_max_segment_size` — set max segment size for `blk_rq_map_sg`

## Synopsis

```
void blk_queue_max_segment_size (struct request_queue * q, unsigned int  
max_size);
```

## Arguments

*q*                    the request queue for the device

*max\_size*    max size of segment in bytes

## Description

Enables a low level driver to set an upper limit on the size of a coalesced segment

## Name

`blk_queue_logical_block_size` — set logical block size for the queue

## Synopsis

```
void blk_queue_logical_block_size (struct request_queue * q, unsigned
short size);
```

## Arguments

*q*        the request queue for the device

*size*    the logical block size, in bytes

## Description

This should be set to the lowest possible block size that the storage device can address. The default of 512 covers most hardware.

## Name

`blk_queue_physical_block_size` — set physical block size for the queue

## Synopsis

```
void blk_queue_physical_block_size (struct request_queue * q, unsigned  
int size);
```

## Arguments

*q*        the request queue for the device

*size*    the physical block size, in bytes

## Description

This should be set to the lowest possible sector size that the hardware can operate on without reverting to read-modify-write operations.

## Name

`blk_queue_alignment_offset` — set physical block alignment offset

## Synopsis

```
void blk_queue_alignment_offset (struct request_queue * q, unsigned int  
offset);
```

## Arguments

*q*            the request queue for the device

*offset*      alignment offset in bytes

## Description

Some devices are naturally misaligned to compensate for things like the legacy DOS partition table 63-sector offset. Low-level drivers should call this function for devices whose first sector is not naturally aligned.

## Name

`blk_limits_io_min` — set minimum request size for a device

## Synopsis

```
void blk_limits_io_min (struct queue_limits * limits, unsigned int min);
```

## Arguments

*limits*    the queue limits

*min*       smallest I/O size in bytes

## Description

Some devices have an internal block size bigger than the reported hardware sector size. This function can be used to signal the smallest I/O the device can perform without incurring a performance penalty.

## Name

`blk_queue_io_min` — set minimum request size for the queue

## Synopsis

```
void blk_queue_io_min (struct request_queue * q, unsigned int min);
```

## Arguments

*q*      the request queue for the device

*min*    smallest I/O size in bytes

## Description

Storage devices may report a granularity or preferred minimum I/O size which is the smallest request the device can perform without incurring a performance penalty. For disk drives this is often the physical block size. For RAID arrays it is often the stripe chunk size. A properly aligned multiple of `minimum_io_size` is the preferred request size for workloads where a high number of I/O operations is desired.

## Name

`blk_limits_io_opt` — set optimal request size for a device

## Synopsis

```
void blk_limits_io_opt (struct queue_limits * limits, unsigned int opt);
```

## Arguments

*limits*    the queue limits

*opt*       smallest I/O size in bytes

## Description

Storage devices may report an optimal I/O size, which is the device's preferred unit for sustained I/O. This is rarely reported for disk drives. For RAID arrays it is usually the stripe width or the internal track size. A properly aligned multiple of `optimal_io_size` is the preferred request size for workloads where sustained throughput is desired.

## Name

`blk_queue_io_opt` — set optimal request size for the queue

## Synopsis

```
void blk_queue_io_opt (struct request_queue * q, unsigned int opt);
```

## Arguments

*q*      the request queue for the device

*opt*    optimal request size in bytes

## Description

Storage devices may report an optimal I/O size, which is the device's preferred unit for sustained I/O. This is rarely reported for disk drives. For RAID arrays it is usually the stripe width or the internal track size. A properly aligned multiple of `optimal_io_size` is the preferred request size for workloads where sustained throughput is desired.



## Name

`blk_queue_stack_limits` — inherit underlying queue limits for stacked drivers

## Synopsis

```
void blk_queue_stack_limits (struct request_queue * t, struct
request_queue * b);
```

## Arguments

*t* the stacking driver (top)

*b* the underlying device (bottom)

## Name

`blk_stack_limits` — adjust `queue_limits` for stacked devices

## Synopsis

```
int blk_stack_limits (struct queue_limits * t, struct queue_limits *  
b, sector_t start);
```

## Arguments

*t*            the stacking driver limits (top device)

*b*            the underlying queue limits (bottom, component device)

*start*       first data sector within component device

## Description

This function is used by stacking drivers like MD and DM to ensure that all component devices have compatible block sizes and alignments. The stacking driver must provide a `queue_limits` struct (top) and then iteratively call the stacking function for all component (bottom) devices. The stacking function will attempt to combine the values and ensure proper alignment.

Returns 0 if the top and bottom `queue_limits` are compatible. The top device's block sizes and alignment offsets may be adjusted to ensure alignment with the bottom device. If no compatible sizes and alignments exist, -1 is returned and the resulting top `queue_limits` will have the `misaligned` flag set to indicate that the `alignment_offset` is undefined.

## Name

`bdev_stack_limits` — adjust queue limits for stacked drivers

## Synopsis

```
int bdev_stack_limits (struct queue_limits * t, struct block_device *  
bdev, sector_t start);
```

## Arguments

*t*            the stacking driver limits (top device)

*bdev*        the component block\_device (bottom)

*start*       first data sector within component device

## Description

Merges queue limits for a top device and a block\_device. Returns 0 if alignment didn't change. Returns -1 if adding the bottom device caused misalignment.

## Name

`disk_stack_limits` — adjust queue limits for stacked drivers

## Synopsis

```
void disk_stack_limits (struct gendisk * disk, struct block_device *  
bdev, sector_t offset);
```

## Arguments

*disk*     MD/DM gendisk (top)

*bdev*     the underlying block device (bottom)

*offset*   offset to beginning of data within component device

## Description

Merges the limits for a top level gendisk and a bottom level `block_device`.

## Name

`blk_queue_dma_pad` — set pad mask

## Synopsis

```
void blk_queue_dma_pad (struct request_queue * q, unsigned int mask);
```

## Arguments

*q*        the request queue for the device

*mask*    pad mask

## Description

Set dma pad mask.

Appending pad buffer to a request modifies the last entry of a scatter list such that it includes the pad buffer.

## Name

`blk_queue_update_dma_pad` — update pad mask

## Synopsis

```
void blk_queue_update_dma_pad (struct request_queue * q, unsigned int
mask);
```

## Arguments

*q*        the request queue for the device

*mask*    pad mask

## Description

Update dma pad mask.

Appending pad buffer to a request modifies the last entry of a scatter list such that it includes the pad buffer.

## Name

`blk_queue_dma_drain` — Set up a drain buffer for excess dma.

## Synopsis

```
int blk_queue_dma_drain (struct request_queue * q, dma_drain_needed_fn  
* dma_drain_needed, void * buf, unsigned int size);
```

## Arguments

<i>q</i>	the request queue for the device
<i>dma_drain_needed</i>	fn which returns non-zero if drain is necessary
<i>buf</i>	physically contiguous buffer
<i>size</i>	size of the buffer in bytes

## Description

Some devices have excess DMA problems and can't simply discard (or zero fill) the unwanted piece of the transfer. They have to have a real area of memory to transfer it into. The use case for this is ATAPI devices in DMA mode. If the packet command causes a transfer bigger than the transfer size some HBAs will lock up if there aren't DMA elements to contain the excess transfer. What this API does is adjust the queue so that the buf is always appended silently to the scatterlist.

## Note

This routine adjusts `max_hw_segments` to make room for appending the drain buffer. If you call `blk_queue_max_segments` after calling this routine, you must set the limit to one fewer than your device can support otherwise there won't be room for the drain buffer.

## Name

`blk_queue_segment_boundary` — set boundary rules for segment merging

## Synopsis

```
void blk_queue_segment_boundary (struct request_queue * q, unsigned long  
mask);
```

## Arguments

*q*        the request queue for the device

*mask*    the memory boundary mask



## Name

`blk_queue_virt_boundary` — set boundary rules for bio merging

## Synopsis

```
void blk_queue_virt_boundary (struct request_queue * q, unsigned long  
mask);
```

## Arguments

*q*        the request queue for the device

*mask*    the memory boundary mask

## Name

`blk_queue_dma_alignment` — set dma length and memory alignment

## Synopsis

```
void blk_queue_dma_alignment (struct request_queue * q, int mask);
```

## Arguments

*q*        the request queue for the device

*mask*    alignment mask

## description

set required memory and length alignment for direct dma transactions. this is used when building direct io requests for the queue.

## Name

`blk_queue_update_dma_alignment` — update dma length and memory alignment

## Synopsis

```
void blk_queue_update_dma_alignment (struct request_queue * q, int
mask);
```

## Arguments

*q* the request queue for the device

*mask* alignment mask

## description

update required memory and length alignment for direct dma transactions. If the requested alignment is larger than the current alignment, then the current queue alignment is updated to the new value, otherwise it is left alone. The design of this is to allow multiple objects (driver, device, transport etc) to set their respective alignments without having them interfere.

## Name

`blk_queue_flush` — configure queue's cache flush capability

## Synopsis

```
void blk_queue_flush (struct request_queue * q, unsigned int flush);
```

## Arguments

*q*            the request queue for the device

*flush*    0, REQ\_FLUSH or REQ\_FLUSH | REQ\_FUA

## Description

Tell block layer cache flush capability of *q*. If it supports flushing, REQ\_FLUSH should be set. If it supports bypassing write cache for individual writes, REQ\_FUA should be set.

## Name

`blk_execute_rq_nowait` — insert a request into queue for execution

## Synopsis

```
void blk_execute_rq_nowait (struct request_queue * q, struct gendisk *  
bd_disk, struct request * rq, int at_head, rq_end_io_fn * done);
```

## Arguments

<i>q</i>	queue to insert the request in
<i>bd_disk</i>	matching gendisk
<i>rq</i>	request to insert
<i>at_head</i>	insert request at head or tail of queue
<i>done</i>	I/O completion handler

## Description

Insert a fully prepared request at the back of the I/O scheduler queue for execution. Don't wait for completion.

## Note

This function will invoke *done* directly if the queue is dead.

## Name

`blk_execute_rq` — insert a request into queue for execution

## Synopsis

```
int blk_execute_rq (struct request_queue * q, struct gendisk * bd_disk,  
struct request * rq, int at_head);
```

## Arguments

<i>q</i>	queue to insert the request in
<i>bd_disk</i>	matching gendisk
<i>rq</i>	request to insert
<i>at_head</i>	insert request at head or tail of queue

## Description

Insert a fully prepared request at the back of the I/O scheduler queue for execution and wait for completion.

## Name

`blkdev_issue_flush` — queue a flush

## Synopsis

```
int blkdev_issue_flush (struct block_device * bdev, gfp_t gfp_mask,  
sector_t * error_sector);
```

## Arguments

<i>bdev</i>	blockdev to issue flush for
<i>gfp_mask</i>	memory allocation flags (for <code>bio_alloc</code> )
<i>error_sector</i>	error sector

## Description

Issue a flush for the block device in question. Caller can supply room for storing the error offset in case of a flush error, if they wish to. If `WAIT` flag is not passed then caller may check only what request was pushed in some internal queue for later handling.

## Name

`blkdev_issue_discard` — queue a discard

## Synopsis

```
int blkdev_issue_discard (struct block_device * bdev, sector_t sector,  
sector_t nr_sects, gfp_t gfp_mask, unsigned long flags);
```

## Arguments

<i>bdev</i>	blockdev to issue discard for
<i>sector</i>	start sector
<i>nr_sects</i>	number of sectors to discard
<i>gfp_mask</i>	memory allocation flags (for <code>bio_alloc</code> )
<i>flags</i>	<code>BLKDEV_IFL_*</code> flags to control behaviour

## Description

Issue a discard request for the sectors in question.



## Name

`blkdev_issue_write_same` — queue a write same operation

## Synopsis

```
int blkdev_issue_write_same (struct block_device * bdev, sector_t
sector, sector_t nr_sects, gfp_t gfp_mask, struct page * page);
```

## Arguments

<i>bdev</i>	target blockdev
<i>sector</i>	start sector
<i>nr_sects</i>	number of sectors to write
<i>gfp_mask</i>	memory allocation flags (for <code>bio_alloc</code> )
<i>page</i>	page containing data to write

## Description

Issue a write same request for the sectors in question.

## Name

`blkdev_issue_zeroout` — zero-fill a block range

## Synopsis

```
int blkdev_issue_zeroout (struct block_device * bdev, sector_t sector,
sector_t nr_sects, gfp_t gfp_mask, bool discard);
```

## Arguments

<i>bdev</i>	blockdev to write
<i>sector</i>	start sector
<i>nr_sects</i>	number of sectors to write
<i>gfp_mask</i>	memory allocation flags (for <code>bio_alloc</code> )
<i>discard</i>	whether to discard the block range

## Description

Zero-fill a block range. If the discard flag is set and the block device guarantees that subsequent READ operations to the block range in question will return zeroes, the blocks will be discarded. Should the discard request fail, if the discard flag is not set, or if `discard_zeroes_data` is not supported, this function will resort to zeroing the blocks manually, thus provisioning (allocating, anchoring) them. If the block device supports the WRITE SAME command `blkdev_issue_zeroout` will use it to optimize the process of clearing the block range. Otherwise the zeroing will be performed using regular WRITE calls.

## Name

`blk_queue_find_tag` — find a request by its tag and queue

## Synopsis

```
struct request * blk_queue_find_tag (struct request_queue * q, int tag);
```

## Arguments

*q*      The request queue for the device

*tag*    The tag of the request

## Notes

Should be used when a device returns a tag and you want to match it with a request.

no locks need be held.

## Name

`blk_free_tags` — release a given set of tag maintenance info

## Synopsis

```
void blk_free_tags (struct blk_queue_tag * bqt);
```

## Arguments

*bqt* the tag map to free

## Description

Drop the reference count on *bqt* and frees it when the last reference is dropped.

## Name

`blk_queue_free_tags` — release tag maintenance info

## Synopsis

```
void blk_queue_free_tags (struct request_queue * q);
```

## Arguments

*q* the request queue for the device

## Notes

This is used to disable tagged queuing to a device, yet leave queue in function.

## Name

`blk_init_tags` — initialize the tag info for an external tag map

## Synopsis

```
struct blk_queue_tag * blk_init_tags (int depth, int alloc_policy);
```

## Arguments

*depth*                    the maximum queue depth supported

*alloc\_policy*    tag allocation policy

## Name

`blk_queue_init_tags` — initialize the queue tag info

## Synopsis

```
int blk_queue_init_tags (struct request_queue * q, int depth, struct
blk_queue_tag * tags, int alloc_policy);
```

## Arguments

<i>q</i>	the request queue for the device
<i>depth</i>	the maximum queue depth supported
<i>tags</i>	the tag to use
<i>alloc_policy</i>	tag allocation policy

## Description

Queue lock must be held here if the function is called to resize an existing map.

## Name

`blk_queue_resize_tags` — change the queueing depth

## Synopsis

```
int blk_queue_resize_tags (struct request_queue * q, int new_depth);
```

## Arguments

*q*                    the request queue for the device

*new\_depth*    the new max command queueing depth

## Notes

Must be called with the queue lock held.



## Name

`blk_queue_end_tag` — end tag operations for a request

## Synopsis

```
void blk_queue_end_tag (struct request_queue * q, struct request * rq);
```

## Arguments

*q* the request queue for the device

*rq* the request that has completed

## Description

Typically called when `end_that_request_first` returns 0, meaning all transfers have been done for a request. It's important to call this function before `end_that_request_last`, as that will put the request back on the free list thus corrupting the internal tag list.

## Notes

queue lock must be held.

## Name

`blk_queue_start_tag` — find a free tag and assign it

## Synopsis

```
int blk_queue_start_tag (struct request_queue * q, struct request * rq);
```

## Arguments

*q* the request queue for the device

*rq* the block request that needs tagging

## Description

This can either be used as a stand-alone helper, or possibly be assigned as the queue `prep_rq_fn` (in which case `struct request` automatically gets a tag assigned). Note that this function assumes that any type of request can be queued! if this is not true for your device, you must check the request type before calling this function. The request will also be removed from the request queue, so it's the drivers responsibility to readd it if it should need to be restarted for some reason.

## Notes

queue lock must be held.

## Name

`blk_queue_invalidate_tags` — invalidate all pending tags

## Synopsis

```
void blk_queue_invalidate_tags (struct request_queue * q);
```

## Arguments

*q* the request queue for the device

## Description

Hardware conditions may dictate a need to stop all pending requests. In this case, we will safely clear the block side of the tag queue and readd all requests to the request queue in the right order.

## Notes

queue lock must be held.

## Name

`__blk_queue_free_tags` — release tag maintenance info

## Synopsis

```
void __blk_queue_free_tags (struct request_queue * q);
```

## Arguments

*q* the request queue for the device

## Notes

`blk_cleanup_queue` will take care of calling this function, if tagging has been used. So there's no need to call this directly.

## Name

`blk_rq_count_integrity_sg` — Count number of integrity scatterlist elements

## Synopsis

```
int blk_rq_count_integrity_sg (struct request_queue * q, struct bio *  
bio);
```

## Arguments

*q*     request queue

*bio*   bio with integrity metadata attached

## Description

Returns the number of elements required in a scatterlist corresponding to the integrity metadata in a bio.

## Name

`blk_rq_map_integrity_sg` — Map integrity metadata into a scatterlist

## Synopsis

```
int blk_rq_map_integrity_sg (struct request_queue * q, struct bio * bio,  
struct scatterlist * sglist);
```

## Arguments

<i>q</i>	request queue
<i>bio</i>	bio with integrity metadata attached
<i>sglist</i>	target scatterlist

## Description

Map the integrity vectors in request into a scatterlist. The scatterlist must be big enough to hold all elements. I.e. sized using `blk_rq_count_integrity_sg`.

## Name

`blk_integrity_compare` — Compare integrity profile of two disks

## Synopsis

```
int blk_integrity_compare (struct gendisk * gd1, struct gendisk * gd2);
```

## Arguments

*gd1*    Disk to compare

*gd2*    Disk to compare

## Description

Meta-devices like DM and MD need to verify that all sub-devices use the same integrity format before advertising to upper layers that they can send/receive integrity metadata. This function can be used to check whether two gendisk devices have compatible integrity formats.

## Name

`blk_integrity_register` — Register a gendisk as being integrity-capable

## Synopsis

```
void blk_integrity_register (struct gendisk * disk, struct blk_integrity
* template);
```

## Arguments

*disk*            struct gendisk pointer to make integrity-aware

*template*       block integrity profile to register

## Description

When a device needs to advertise itself as being able to send/receive integrity metadata it must use this function to register the capability with the block layer. The template is a `blk_integrity` struct with values appropriate for the underlying hardware. See [Documentation/block/data-integrity.txt](#).



## Name

`blk_integrity_unregister` — Unregister block integrity profile

## Synopsis

```
void blk_integrity_unregister (struct gendisk * disk);
```

## Arguments

*disk* disk whose integrity profile to unregister

## Description

This function unregisters the integrity capability from a block device.

## Name

`blk_trace_ioctl` — handle the ioctls associated with tracing

## Synopsis

```
int blk_trace_ioctl (struct block_device * bdev, unsigned cmd, char
__user * arg);
```

## Arguments

*bdev* the block device

*cmd* the ioctl cmd

*arg* the argument data, if any

## Name

`blk_trace_shutdown` — stop and cleanup trace structures

## Synopsis

```
void blk_trace_shutdown (struct request_queue * q);
```

## Arguments

*q* the request queue associated with the device

## Name

`blk_add_trace_rq` — Add a trace for a request oriented action

## Synopsis

```
void blk_add_trace_rq (struct request_queue * q, struct request * rq,  
unsigned int nr_bytes, u32 what);
```

## Arguments

<i>q</i>	queue the io is for
<i>rq</i>	the source request
<i>nr_bytes</i>	number of completed bytes
<i>what</i>	the action

## Description

Records an action against a request. Will log the bio offset + size.

## Name

`blk_add_trace_bio` — Add a trace for a bio oriented action

## Synopsis

```
void blk_add_trace_bio (struct request_queue * q, struct bio * bio, u32  
what, int error);
```

## Arguments

*q*           queue the io is for

*bio*        the source bio

*what*      the action

*error*     error, if any

## Description

Records an action against a bio. Will log the bio offset + size.

## Name

`blk_add_trace_bio_remap` — Add a trace for a bio-remap operation

## Synopsis

```
void blk_add_trace_bio_remap (void * ignore, struct request_queue * q,  
struct bio * bio, dev_t dev, sector_t from);
```

## Arguments

*ignore*    trace callback data parameter (not used)

*q*        queue the io is for

*bio*       the source bio

*dev*       target device

*from*      source sector

## Description

Device mapper or raid target sometimes need to split a bio because it spans a stripe (or similar). Add a trace for that action.

## Name

`blk_add_trace_rq_remap` — Add a trace for a request-remap operation

## Synopsis

```
void blk_add_trace_rq_remap (void * ignore, struct request_queue * q,  
struct request * rq, dev_t dev, sector_t from);
```

## Arguments

*ignore*    trace callback data parameter (not used)

*q*            queue the io is for

*rq*            the source request

*dev*          target device

*from*        source sector

## Description

Device mapper remaps request to other devices. Add a trace for that action.

## Name

`blk_mangle_minor` — scatter minor numbers apart

## Synopsis

```
int blk_mangle_minor (int minor);
```

## Arguments

*minor*    minor number to mangle

## Description

Scatter consecutively allocated *minor* number apart if `MANGLE_DEVT` is enabled. Mangling twice gives the original value.

## RETURNS

Mangled value.

## CONTEXT

Don't care.



## Name

`blk_alloc_dev_t` — allocate a `dev_t` for a partition

## Synopsis

```
int blk_alloc_dev_t (struct hd_struct * part, dev_t * devt);
```

## Arguments

*part* partition to allocate `dev_t` for

*devt* out parameter for resulting `dev_t`

## Description

Allocate a `dev_t` for block device.

## RETURNS

0 on success, allocated `dev_t` is returned in `*devt`. -errno on failure.

## CONTEXT

Might sleep.

## Name

`blk_free_dev_t` — free a `dev_t`

## Synopsis

```
void blk_free_dev_t (dev_t devt);
```

## Arguments

*devt*    `dev_t` to free

## Description

Free *devt* which was allocated using `blk_alloc_dev_t`.

## CONTEXT

Might sleep.

## Name

`disk_replace_part_tbl` — replace `disk->part_tbl` in RCU-safe way

## Synopsis

```
void disk_replace_part_tbl (struct gendisk * disk, struct disk_part_tbl  
* new_ptbl);
```

## Arguments

*disk*            disk to replace `part_tbl` for

*new\_ptbl*    new `part_tbl` to install

## Description

Replace `disk->part_tbl` with *new\_ptbl* in RCU-safe way. The original `ptbl` is freed using RCU callback.

## LOCKING

Matching `bd_mutex` locked.

## Name

`disk_expand_part_tbl` — expand `disk->part_tbl`

## Synopsis

```
int disk_expand_part_tbl (struct gendisk * disk, int partno);
```

## Arguments

*disk*      disk to expand `part_tbl` for

*partno*    expand such that this `partno` can fit in

## Description

Expand `disk->part_tbl` such that *partno* can fit in. `disk->part_tbl` uses RCU to allow unlocked dereferencing for stats and other stuff.

## LOCKING

Matching `bd_mutex` locked, might sleep.

## RETURNS

0 on success, `-errno` on failure.

## Name

`disk_block_events` — block and flush disk event checking

## Synopsis

```
void disk_block_events (struct gendisk * disk);
```

## Arguments

*disk* disk to block events for

## Description

On return from this function, it is guaranteed that event checking isn't in progress and won't happen until unblocked by `disk_unblock_events`. Events blocking is counted and the actual unblocking happens after the matching number of unblocks are done.

Note that this intentionally does not block event checking from `disk_clear_events`.

## CONTEXT

Might sleep.

## Name

`disk_unblock_events` — unblock disk event checking

## Synopsis

```
void disk_unblock_events (struct gendisk * disk);
```

## Arguments

*disk* disk to unblock events for

## Description

Undo `disk_block_events`. When the block count reaches zero, it starts events polling if configured.

## CONTEXT

Don't care. Safe to call from irq context.

## Name

`disk_flush_events` — schedule immediate event checking and flushing

## Synopsis

```
void disk_flush_events (struct gendisk * disk, unsigned int mask);
```

## Arguments

*disk* disk to check and flush events for

*mask* events to flush

## Description

Schedule immediate event checking on *disk* if not blocked. Events in *mask* are scheduled to be cleared from the driver. Note that this doesn't clear the events from *disk->ev*.

## CONTEXT

If *mask* is non-zero must be called with `bdev->bd_mutex` held.

## Name

`disk_clear_events` — synchronously check, clear and return pending events

## Synopsis

```
unsigned int disk_clear_events (struct gendisk * disk, unsigned int
mask);
```

## Arguments

*disk* disk to fetch and clear events from

*mask* mask of events to be fetched and cleared

## Description

Disk events are synchronously checked and pending events in *mask* are cleared and returned. This ignores the block count.

## CONTEXT

Might sleep.



## Name

`disk_get_part` — get partition

## Synopsis

```
struct hd_struct * disk_get_part (struct gendisk * disk, int partno);
```

## Arguments

*disk*      disk to look partition from

*partno*    partition number

## Description

Look for partition *partno* from *disk*. If found, increment reference count and return it.

## CONTEXT

Don't care.

## RETURNS

Pointer to the found partition on success, NULL if not found.

## Name

`disk_part_iter_init` — initialize partition iterator

## Synopsis

```
void disk_part_iter_init (struct disk_part_iter * piter, struct gendisk  
* disk, unsigned int flags);
```

## Arguments

*piter* iterator to initialize

*disk* disk to iterate over

*flags* DISK\_PITER\_\* flags

## Description

Initialize *piter* so that it iterates over partitions of *disk*.

## CONTEXT

Don't care.

## Name

`disk_part_iter_next` — proceed iterator to the next partition and return it

## Synopsis

```
struct hd_struct * disk_part_iter_next (struct disk_part_iter * piter);
```

## Arguments

*piter* iterator of interest

## Description

Proceed *piter* to the next partition and return it.

## CONTEXT

Don't care.

## Name

`disk_part_iter_exit` — finish up partition iteration

## Synopsis

```
void disk_part_iter_exit (struct disk_part_iter * piter);
```

## Arguments

*piter*   iter of interest

## Description

Called when iteration is over. Cleans up *piter*.

## CONTEXT

Don't care.

## Name

`disk_map_sector_rcu` — map sector to partition

## Synopsis

```
struct hd_struct * disk_map_sector_rcu (struct gendisk * disk, sector_t  
sector);
```

## Arguments

*disk*      gendisk of interest

*sector*    sector to map

## Description

Find out which partition *sector* maps to on *disk*. This is primarily used for stats accounting.

## CONTEXT

RCU read locked. The returned partition pointer is valid only while preemption is disabled.

## RETURNS

Found partition on success, `part0` is returned if no partition matches

## Name

register\_blkdev — register a new block device

## Synopsis

```
int register_blkdev (unsigned int major, const char * name);
```

## Arguments

*major* the requested major device number [1..255]. If *major*=0, try to allocate any unused major number.

*name* the name of the new block device as a zero terminated string

## Description

The *name* must be unique within the system.

The return value depends on the *major* input parameter. - if a major device number was requested in range [1..255] then the function returns zero on success, or a negative error code - if any unused major number was requested with *major*=0 parameter then the return value is the allocated major number in range [1..255] or a negative error code otherwise

## Name

`add_disk` — add partitioning information to kernel list

## Synopsis

```
void add_disk (struct gendisk * disk);
```

## Arguments

*disk* per-device partitioning information

## Description

This function registers the partitioning information in *disk* with the kernel.

## FIXME

error handling

## Name

`get_gendisk` — get partitioning information for a given device

## Synopsis

```
struct gendisk * get_gendisk (dev_t devt, int * partno);
```

## Arguments

*devt*      device to get partitioning information for

*partno*    returned partition index

## Description

This function gets the structure containing partitioning information for the given device *devt*.



## Name

`bdget_disk` — do bdget by gendisk and partition number

## Synopsis

```
struct block_device * bdget_disk (struct gendisk * disk, int partno);
```

## Arguments

*disk*      gendisk of interest

*partno*    partition number

## Description

Find partition *partno* from *disk*, do bdget on it.

## CONTEXT

Don't care.

## RETURNS

Resulting `block_device` on success, NULL on failure.

---

## Chapter 15. Char devices

## Name

`register_chrdev_region` — register a range of device numbers

## Synopsis

```
int register_chrdev_region (dev_t from, unsigned count, const char *  
name);
```

## Arguments

*from*    the first in the desired range of device numbers; must include the major number.

*count*   the number of consecutive device numbers required

*name*    the name of the device or driver.

## Description

Return value is zero on success, a negative error code on failure.

## Name

`alloc_chrdev_region` — register a range of char device numbers

## Synopsis

```
int alloc_chrdev_region (dev_t * dev, unsigned baseminor, unsigned
count, const char * name);
```

## Arguments

*dev*                output parameter for first assigned number

*baseminor*       first of the requested range of minor numbers

*count*            the number of minor numbers required

*name*             the name of the associated device or driver

## Description

Allocates a range of char device numbers. The major number will be chosen dynamically, and returned (along with the first minor number) in *dev*. Returns zero or a negative error code.

## Name

`__register_chrdev` — create and register a cdev occupying a range of minors

## Synopsis

```
int __register_chrdev (unsigned int major, unsigned int baseminor,
unsigned int count, const char * name, const struct file_operations
* fops);
```

## Arguments

<i>major</i>	major device number or 0 for dynamic allocation
<i>baseminor</i>	first of the requested range of minor numbers
<i>count</i>	the number of minor numbers required
<i>name</i>	name of this range of devices
<i>fops</i>	file operations associated with this devices

## Description

If *major* == 0 this functions will dynamically allocate a major and return its number.

If *major* > 0 this function will attempt to reserve a device with the given major number and will return zero on success.

Returns a -ve errno on failure.

The name of this device has nothing to do with the name of the device in /dev. It only helps to keep track of the different owners of devices. If your module name has only one type of devices it's ok to use e.g. the name of the module here.

## Name

`unregister_chrdev_region` — unregister a range of device numbers

## Synopsis

```
void unregister_chrdev_region (dev_t from, unsigned count);
```

## Arguments

*from*     the first in the range of numbers to unregister

*count*    the number of device numbers to unregister

## Description

This function will unregister a range of *count* device numbers, starting with *from*. The caller should normally be the one who allocated those numbers in the first place...

## Name

`__unregister_chrdev` — unregister and destroy a cdev

## Synopsis

```
void __unregister_chrdev (unsigned int major, unsigned int baseminor,  
unsigned int count, const char * name);
```

## Arguments

<i>major</i>	major device number
<i>baseminor</i>	first of the range of minor numbers
<i>count</i>	the number of minor numbers this cdev is occupying
<i>name</i>	name of this range of devices

## Description

Unregister and destroy the cdev occupying the region described by *major*, *baseminor* and *count*. This function undoes what `__register_chrdev` did.

## Name

`cdev_add` — add a char device to the system

## Synopsis

```
int cdev_add (struct cdev * p, dev_t dev, unsigned count);
```

## Arguments

*p*            the cdev structure for the device

*dev*        the first device number for which this device is responsible

*count*    the number of consecutive minor numbers corresponding to this device

## Description

`cdev_add` adds the device represented by *p* to the system, making it live immediately. A negative error code is returned on failure.



## Name

`cdev_del` — remove a cdev from the system

## Synopsis

```
void cdev_del (struct cdev * p);
```

## Arguments

*p* the cdev structure to be removed

## Description

`cdev_del` removes *p* from the system, possibly freeing the structure itself.

## Name

`cdev_alloc` — allocate a `cdev` structure

## Synopsis

```
struct cdev * cdev_alloc ( void );
```

## Arguments

*void* no arguments

## Description

Allocates and returns a `cdev` structure, or `NULL` on failure.

## Name

`cdev_init` — initialize a `cdev` structure

## Synopsis

```
void cdev_init (struct cdev * cdev, const struct file_operations * fops);
```

## Arguments

*cdev* the structure to initialize

*fops* the `file_operations` for this device

## Description

Initializes *cdev*, remembering *fops*, making it ready to add to the system with `cdev_add`.

---

# Chapter 16. Miscellaneous Devices

## 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. By default, an `open` syscall to the device sets `file->private_data` to point to the structure. Drivers don't need `open` in fops for this.

A zero is returned on success and a negative `errno` code for failure.

## Name

`misc_deregister` — unregister a miscellaneous device

## Synopsis

```
void misc_deregister (struct miscdevice * misc);
```

## Arguments

*misc* device to unregister

## Description

Unregister a miscellaneous device that was previously successfully registered with `misc_register`.

---

# Chapter 17. Clock Framework

The clock framework defines programming interfaces to support software management of the system clock tree. This framework is widely used with System-On-Chip (SOC) platforms to support power management and various devices which may need custom clock rates. Note that these "clocks" don't relate to timekeeping or real time clocks (RTCs), each of which have separate frameworks. These struct clk instances may be used to manage for example a 96 MHz signal that is used to shift bits into and out of peripherals or busses, or otherwise trigger synchronous state machine transitions in system hardware.

Power management is supported by explicit software clock gating: unused clocks are disabled, so the system doesn't waste power changing the state of transistors that aren't in active use. On some systems this may be backed by hardware clock gating, where clocks are gated without being disabled in software. Sections of chips that are powered but not clocked may be able to retain their last state. This low power state is often called a *retention mode*. This mode still incurs leakage currents, especially with finer circuit geometries, but for CMOS circuits power is mostly used by clocked state changes.

Power-aware drivers only enable their clocks when the device they manage is in active use. Also, system sleep states often differ according to which clock domains are active: while a "standby" state may allow wakeup from several active domains, a "mem" (suspend-to-RAM) state may require a more wholesale shutdown of clocks derived from higher speed PLLs and oscillators, limiting the number of possible wakeup event sources. A driver's suspend method may need to be aware of system-specific clock constraints on the target sleep state.

Some platforms support programmable clock generators. These can be used by external chips of various kinds, such as other CPUs, multimedia codecs, and devices with strict requirements for interface clocking.

## Name

struct `clk_notifier` — associate a `clk` with a notifier

## Synopsis

```
struct clk_notifier {  
    struct clk * clk;  
    struct srcu_notifier_head notifier_head;  
    struct list_head node;  
};
```

## Members

<code>clk</code>	struct <code>clk *</code> to associate the notifier with
<code>notifier_head</code>	a <code>blocking_notifier_head</code> for this <code>clk</code>
<code>node</code>	linked list pointers

## Description

A list of struct `clk_notifier` is maintained by the notifier code. An entry is created whenever code registers the first notifier on a particular `clk`. Future notifiers on that `clk` are added to the `notifier_head`.



## Name

struct clk\_notifier\_data — rate data to pass to the notifier callback

## Synopsis

```
struct clk_notifier_data {  
    struct clk * clk;  
    unsigned long old_rate;  
    unsigned long new_rate;  
};
```

## Members

clk	struct clk * being changed
old_rate	previous rate of this clk
new_rate	new rate of this clk

## Description

For a pre-notifier, old\_rate is the clk's rate before this rate change, and new\_rate is what the rate will be in the future. For a post-notifier, old\_rate and new\_rate are both set to the clk's current rate (this was done to optimize the implementation).

## Name

`clk_notifier_register` — change notifier callback

## Synopsis

```
int clk_notifier_register (struct clk * clk, struct notifier_block *  
nb);
```

## Arguments

*clk*   clock whose rate we are interested in

*nb*   notifier block with callback function pointer

## ProTip

debugging across notifier chains can be frustrating. Make sure that your notifier callback function prints a nice big warning in case of failure.

## Name

`clk_notifier_unregister` — change notifier callback

## Synopsis

```
int clk_notifier_unregister (struct clk * clk, struct notifier_block  
* nb);
```

## Arguments

*clk* clock whose rate we are no longer interested in

*nb* notifier block which will be unregistered

## Name

`clk_get_accuracy` — obtain the clock accuracy in ppb (parts per billion) for a clock source.

## Synopsis

```
long clk_get_accuracy (struct clk * clk);
```

## Arguments

*clk* clock source

## Description

This gets the clock source accuracy expressed in ppb. A perfect clock returns 0.

## Name

`clk_set_phase` — adjust the phase shift of a clock signal

## Synopsis

```
int clk_set_phase (struct clk * clk, int degrees);
```

## Arguments

*clk*            clock signal source

*degrees*    number of degrees the signal is shifted

## Description

Shifts the phase of a clock signal by the specified degrees. Returns 0 on success, -EERROR otherwise.

## Name

`clk_get_phase` — return the phase shift of a clock signal

## Synopsis

```
int clk_get_phase (struct clk * clk);
```

## Arguments

*clk* clock signal source

## Description

Returns the phase shift of a clock node in degrees, otherwise returns -EERROR.

## Name

`clk_is_match` — check if two clk's point to the same hardware clock

## Synopsis

```
bool clk_is_match (const struct clk * p, const struct clk * q);
```

## Arguments

*p* clk compared against *q*

*q* clk compared against *p*

## Description

Returns true if the two struct clk pointers both point to the same hardware clock node. Put differently, returns true if struct clk \**p* and struct clk \**q* share the same struct clk\_core object.

Returns false otherwise. Note that two NULL clks are treated as matching.

## Name

`clk_prepare` — prepare a clock source

## Synopsis

```
int clk_prepare (struct clk * clk);
```

## Arguments

*clk* clock source

## Description

This prepares the clock source for use.

Must not be called from within atomic context.



## Name

`clk_unprepare` — undo preparation of a clock source

## Synopsis

```
void clk_unprepare (struct clk * clk);
```

## Arguments

*clk* clock source

## Description

This undoes a previously prepared clock. The caller must balance the number of prepare and unprepare calls.

Must not be called from within atomic context.

## Name

`clk_get` — lookup and obtain a reference to a clock producer.

## Synopsis

```
struct clk * clk_get (struct device * dev, const char * id);
```

## Arguments

*dev*    device for clock “consumer”

*id*     clock consumer ID

## Description

Returns a struct clk corresponding to the clock producer, or valid IS\_ERR condition containing errno. The implementation uses *dev* and *id* to determine the clock consumer, and thereby the clock producer. (IOW, *id* may be identical strings, but `clk_get` may return different clock producers depending on *dev*.)

Drivers must assume that the clock source is not enabled.

`clk_get` should not be called from within interrupt context.

## Name

`devm_clk_get` — lookup and obtain a managed reference to a clock producer.

## Synopsis

```
struct clk * devm_clk_get (struct device * dev, const char * id);
```

## Arguments

*dev*    device for clock “consumer”

*id*    clock consumer ID

## Description

Returns a struct clk corresponding to the clock producer, or valid `IS_ERR` condition containing `errno`. The implementation uses *dev* and *id* to determine the clock consumer, and thereby the clock producer. (IOW, *id* may be identical strings, but `clk_get` may return different clock producers depending on *dev*.)

Drivers must assume that the clock source is not enabled.

`devm_clk_get` should not be called from within interrupt context.

The clock will automatically be freed when the device is unbound from the bus.

## Name

`clk_enable` — inform the system when the clock source should be running.

## Synopsis

```
int clk_enable (struct clk * clk);
```

## Arguments

*clk* clock source

## Description

If the clock can not be enabled/disabled, this should return success.

May be called from atomic contexts.

Returns success (0) or negative errno.

## Name

`clk_disable` — inform the system when the clock source is no longer required.

## Synopsis

```
void clk_disable (struct clk * clk);
```

## Arguments

*clk* clock source

## Description

Inform the system that a clock source is no longer required by a driver and may be shut down.

May be called from atomic contexts.

## Implementation detail

if the clock source is shared between multiple drivers, `clk_enable` calls must be balanced by the same number of `clk_disable` calls for the clock source to be disabled.

## Name

`clk_get_rate` — obtain the current clock rate (in Hz) for a clock source. This is only valid once the clock source has been enabled.

## Synopsis

```
unsigned long clk_get_rate (struct clk * clk);
```

## Arguments

*clk* clock source

## Name

`clk_put` — "free" the clock source

## Synopsis

```
void clk_put (struct clk * clk);
```

## Arguments

*clk* clock source

## Note

drivers must ensure that all `clk_enable` calls made on this clock source are balanced by `clk_disable` calls prior to calling this function.

`clk_put` should not be called from within interrupt context.

## Name

`devm_clk_put` — "free" a managed clock source

## Synopsis

```
void devm_clk_put (struct device * dev, struct clk * clk);
```

## Arguments

*dev* device used to acquire the clock

*clk* clock source acquired with `devm_clk_get`

## Note

drivers must ensure that all `clk_enable` calls made on this clock source are balanced by `clk_disable` calls prior to calling this function.

`clk_put` should not be called from within interrupt context.



## Name

`clk_round_rate` — adjust a rate to the exact rate a clock can provide

## Synopsis

```
long clk_round_rate (struct clk * clk, unsigned long rate);
```

## Arguments

*clk*     clock source

*rate*    desired clock rate in Hz

## Description

This answers the question “if I were to pass *rate* to `clk_set_rate`, what clock rate would I end up with?” without changing the hardware in any way. In other words:

```
rate = clk_round_rate(clk, r);
```

## and

```
clk_set_rate(clk, r); rate = clk_get_rate(clk);
```

are equivalent except the former does not modify the clock hardware in any way.

Returns rounded clock rate in Hz, or negative `errno`.

## Name

`clk_set_rate` — set the clock rate for a clock source

## Synopsis

```
int clk_set_rate (struct clk * clk, unsigned long rate);
```

## Arguments

*clk*     clock source

*rate*    desired clock rate in Hz

## Description

Returns success (0) or negative errno.

## Name

`clk_has_parent` — check if a clock is a possible parent for another

## Synopsis

```
bool clk_has_parent (struct clk * clk, struct clk * parent);
```

## Arguments

*clk*        clock source

*parent*    parent clock source

## Description

This function can be used in drivers that need to check that a clock can be the parent of another without actually changing the parent.

Returns true if *parent* is a possible parent for *clk*, false otherwise.

## Name

`clk_set_rate_range` — set a rate range for a clock source

## Synopsis

```
int clk_set_rate_range (struct clk * clk, unsigned long min, unsigned  
long max);
```

## Arguments

*clk* clock source

*min* desired minimum clock rate in Hz, inclusive

*max* desired maximum clock rate in Hz, inclusive

## Description

Returns success (0) or negative errno.

## Name

`clk_set_min_rate` — set a minimum clock rate for a clock source

## Synopsis

```
int clk_set_min_rate (struct clk * clk, unsigned long rate);
```

## Arguments

*clk*    clock source

*rate*   desired minimum clock rate in Hz, inclusive

## Description

Returns success (0) or negative errno.

## Name

`clk_set_max_rate` — set a maximum clock rate for a clock source

## Synopsis

```
int clk_set_max_rate (struct clk * clk, unsigned long rate);
```

## Arguments

*clk*    clock source

*rate*   desired maximum clock rate in Hz, inclusive

## Description

Returns success (0) or negative errno.

## Name

`clk_set_parent` — set the parent clock source for this clock

## Synopsis

```
int clk_set_parent (struct clk * clk, struct clk * parent);
```

## Arguments

*clk*        clock source

*parent*    parent clock source

## Description

Returns success (0) or negative errno.

## Name

`clk_get_parent` — get the parent clock source for this clock

## Synopsis

```
struct clk * clk_get_parent (struct clk * clk);
```

## Arguments

*clk* clock source

## Description

Returns struct clk corresponding to parent clock source, or valid IS\_ERR condition containing errno.



## Name

`clk_get_sys` — get a clock based upon the device name

## Synopsis

```
struct clk * clk_get_sys (const char * dev_id, const char * con_id);
```

## Arguments

*dev\_id* device name

*con\_id* connection ID

## Description

Returns a struct `clk` corresponding to the clock producer, or valid `IS_ERR` condition containing `errno`. The implementation uses *dev\_id* and *con\_id* to determine the clock consumer, and thereby the clock producer. In contrast to `clk_get` this function takes the device name instead of the device itself for identification.

Drivers must assume that the clock source is not enabled.

`clk_get_sys` should not be called from within interrupt context.