

# **Linux Device Drivers**

---

# Linux Device Drivers

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. Driver Basics .....	1
Driver Entry and Exit points .....	1
Atomic and pointer manipulation .....	2
Delaying, scheduling, and timer routines .....	16
Wait queues and Wake events .....	71
High-resolution timers .....	103
Workqueues and Kevents .....	123
Internal Functions .....	152
Kernel objects manipulation .....	189
Kernel utility functions .....	203
Device Resource Management .....	264
2. Device drivers infrastructure .....	288
The Basic Device Driver-Model Structures .....	288
Device Drivers Base .....	301
Device Drivers DMA Management .....	406
Device Drivers Power Management .....	460
Device Drivers ACPI Support .....	466
Device drivers PnP support .....	471
Userspace IO devices .....	483
3. Parallel Port Devices .....	490
parport_yield .....	491
parport_yield_blocking .....	492
parport_wait_event .....	493
parport_wait_peripheral .....	494
parport_negotiate .....	495
parport_write .....	496
parport_read .....	497
parport_set_timeout .....	498
__parport_register_driver .....	499
parport_unregister_driver .....	500
parport_get_port .....	501
parport_put_port .....	502
parport_register_port .....	503
parport_announce_port .....	504
parport_remove_port .....	505
parport_register_device .....	506
parport_unregister_device .....	508
parport_find_number .....	509
parport_find_base .....	510
parport_claim .....	511
parport_claim_or_block .....	512
parport_release .....	513
parport_open .....	514
parport_close .....	515
4. Message-based devices .....	516
Fusion message devices .....	516
5. Sound Devices .....	635
snd_printk .....	636
snd_printd .....	637
snd_BUG .....	638
snd_printd_ratelimit .....	639

snd_BUG_ON .....	640
snd_printdd .....	641
register_sound_special_device .....	642
register_sound_mixer .....	643
register_sound_midi .....	644
register_sound_dsp .....	645
unregister_sound_special .....	646
unregister_sound_mixer .....	647
unregister_sound_midi .....	648
unregister_sound_dsp .....	649
snd_pcm_stream_linked .....	650
snd_pcm_stream_lock_irqsave .....	651
snd_pcm_group_for_each_entry .....	652
snd_pcm_running .....	653
bytes_to_samples .....	654
bytes_to_frames .....	655
samples_to_bytes .....	656
frames_to_bytes .....	657
frame_aligned .....	658
snd_pcm_lib_buffer_bytes .....	659
snd_pcm_lib_period_bytes .....	660
snd_pcm_playback_avail .....	661
snd_pcm_capture_avail .....	662
snd_pcm_playback_hw_avail .....	663
snd_pcm_capture_hw_avail .....	664
snd_pcm_playback_ready .....	665
snd_pcm_capture_ready .....	666
snd_pcm_playback_data .....	667
snd_pcm_playback_empty .....	668
snd_pcm_capture_empty .....	669
snd_pcm_trigger_done .....	670
params_channels .....	671
params_rate .....	672
params_period_size .....	673
params_periods .....	674
params_buffer_size .....	675
params_buffer_bytes .....	676
snd_pcm_hw_constraint_single .....	677
snd_pcm_format_cpu_endian .....	678
snd_pcm_set_runtime_buffer .....	679
snd_pcm_gettime .....	680
snd_pcm_lib_alloc_vmalloc_buffer .....	681
snd_pcm_lib_alloc_vmalloc_32_buffer .....	682
snd_pcm_sgbuf_get_addr .....	683
snd_pcm_sgbuf_get_ptr .....	684
snd_pcm_sgbuf_get_chunk_size .....	685
snd_pcm_mmap_data_open .....	686
snd_pcm_mmap_data_close .....	687
snd_pcm_limit_isa_dma_size .....	688
snd_pcm_stream_str .....	689
snd_pcm_chmap_substream .....	690
pcm_format_to_bits .....	691
snd_pcm_format_name .....	692
snd_pcm_new_stream .....	693

snd_pcm_new .....	694
snd_pcm_new_internal .....	695
snd_pcm_notify .....	696
snd_device_new .....	697
snd_device_disconnect .....	698
snd_device_free .....	699
snd_device_register .....	700
snd_info_get_line .....	701
snd_info_get_str .....	702
snd_info_create_module_entry .....	703
snd_info_create_card_entry .....	704
snd_info_free_entry .....	705
snd_info_register .....	706
snd_rawmidi_receive .....	707
snd_rawmidi_transmit_empty .....	708
__snd_rawmidi_transmit_peek .....	709
snd_rawmidi_transmit_peek .....	710
__snd_rawmidi_transmit_ack .....	711
snd_rawmidi_transmit_ack .....	712
snd_rawmidi_transmit .....	713
snd_rawmidi_new .....	714
snd_rawmidi_set_ops .....	715
snd_request_card .....	716
snd_lookup_minor_data .....	717
snd_register_device .....	718
snd_unregister_device .....	719
copy_to_user_fromio .....	720
copy_from_user_toio .....	721
snd_pcm_lib_preallocate_free_for_all .....	722
snd_pcm_lib_preallocate_pages .....	723
snd_pcm_lib_preallocate_pages_for_all .....	724
snd_pcm_sgbuf_ops_page .....	725
snd_pcm_lib_malloc_pages .....	726
snd_pcm_lib_free_pages .....	727
snd_pcm_lib_free_vmalloc_buffer .....	728
snd_pcm_lib_get_vmalloc_page .....	729
snd_device_initialize .....	730
snd_card_new .....	731
snd_card_disconnect .....	732
snd_card_free_when_closed .....	733
snd_card_free .....	734
snd_card_set_id .....	735
snd_card_add_dev_attr .....	736
snd_card_register .....	737
snd_component_add .....	738
snd_card_file_add .....	739
snd_card_file_remove .....	740
snd_power_wait .....	741
snd_dma_program .....	742
snd_dma_disable .....	743
snd_dma_pointer .....	744
snd_ctl_notify .....	745
snd_ctl_new1 .....	746
snd_ctl_free_one .....	747

snd_ctl_add .....	748
snd_ctl_replace .....	749
snd_ctl_remove .....	750
snd_ctl_remove_id .....	751
snd_ctl_activate_id .....	752
snd_ctl_rename_id .....	753
snd_ctl_find_numid .....	754
snd_ctl_find_id .....	755
snd_ctl_register_ioctl .....	756
snd_ctl_register_ioctl_compat .....	757
snd_ctl_unregister_ioctl .....	758
snd_ctl_unregister_ioctl_compat .....	759
snd_ctl_boolean_mono_info .....	760
snd_ctl_boolean_stereo_info .....	761
snd_ctl_enum_info .....	762
snd_pcm_set_ops .....	763
snd_pcm_set_sync .....	764
snd_interval_refine .....	765
snd_interval_ratnum .....	766
snd_interval_list .....	767
snd_interval_ranges .....	768
snd_pcm_hw_rule_add .....	769
snd_pcm_hw_constraint_mask64 .....	770
snd_pcm_hw_constraint_integer .....	771
snd_pcm_hw_constraint_minmax .....	772
snd_pcm_hw_constraint_list .....	773
snd_pcm_hw_constraint_ranges .....	774
snd_pcm_hw_constraint_ratnums .....	775
snd_pcm_hw_constraint_ratdens .....	776
snd_pcm_hw_constraint_msbits .....	777
snd_pcm_hw_constraint_step .....	778
snd_pcm_hw_constraint_pow2 .....	779
snd_pcm_hw_rule_noresample .....	780
snd_pcm_hw_param_value .....	781
snd_pcm_hw_param_first .....	782
snd_pcm_hw_param_last .....	783
snd_pcm_lib_ioctl .....	784
snd_pcm_period_elapsed .....	785
snd_pcm_add_chmap_ctls .....	786
snd_hwdep_new .....	787
snd_pcm_stream_lock .....	788
snd_pcm_stream_unlock .....	789
snd_pcm_stream_lock_irq .....	790
snd_pcm_stream_unlock_irq .....	791
snd_pcm_stream_unlock_irqrestore .....	792
snd_pcm_stop .....	793
snd_pcm_stop_xrun .....	794
snd_pcm_suspend .....	795
snd_pcm_suspend_all .....	796
snd_pcm_lib_default_mmap .....	797
snd_pcm_lib_mmap_iomem .....	798
snd_malloc_pages .....	799
snd_free_pages .....	800
snd_dma_alloc_pages .....	801

snd_dma_alloc_pages_fallback .....	802
snd_dma_free_pages .....	803
6. Media Devices .....	804
Video2Linux devices .....	804
Digital TV (DVB) devices .....	922
Remote Controller devices .....	972
Media Controller devices .....	979
7. 16x50 UART Driver .....	983
uart_update_timeout .....	984
uart_get_baud_rate .....	985
uart_get_divisor .....	986
uart_console_write .....	987
uart_parse_earlycon .....	988
uart_parse_options .....	989
uart_set_options .....	990
uart_register_driver .....	991
uart_unregister_driver .....	992
uart_add_one_port .....	993
uart_remove_one_port .....	994
uart_handle_dcd_change .....	995
uart_handle_cts_change .....	996
uart_insert_char .....	997
serial8250_get_port .....	998
serial8250_suspend_port .....	999
serial8250_resume_port .....	1000
serial8250_register_8250_port .....	1001
serial8250_unregister_port .....	1002
8. Frame Buffer Library .....	1003
Frame Buffer Memory .....	1003
Frame Buffer Colormap .....	1006
Frame Buffer Video Mode Database .....	1011
Frame Buffer Macintosh Video Mode Database .....	1023
Frame Buffer Fonts .....	1026
9. Input Subsystem .....	1027
Input core .....	1027
Multitouch Library .....	1069
Polled input devices .....	1081
Matrix keyboards/keypads .....	1087
Sparse keymap support .....	1090
10. Serial Peripheral Interface (SPI) .....	1098
struct spi_statistics .....	1099
struct spi_device .....	1100
struct spi_driver .....	1102
spi_unregister_driver .....	1103
module_spi_driver .....	1104
struct spi_master .....	1105
struct spi_transfer .....	1109
struct spi_message .....	1111
spi_message_init_with_transfers .....	1112
spi_write .....	1113
spi_read .....	1114
spi_sync_transfer .....	1115
spi_w8r8 .....	1116
spi_w8r16 .....	1117

spi_w8r16be .....	1118
struct spi_board_info .....	1119
spi_register_board_info .....	1120
__spi_register_driver .....	1121
spi_alloc_device .....	1122
spi_add_device .....	1123
spi_new_device .....	1124
spi_finalize_current_transfer .....	1125
spi_get_next_queued_message .....	1126
spi_finalize_current_message .....	1127
spi_alloc_master .....	1128
spi_register_master .....	1129
devm_spi_register_master .....	1130
spi_unregister_master .....	1131
spi_busnum_to_master .....	1132
spi_setup .....	1133
spi_async .....	1134
spi_async_locked .....	1135
spi_sync .....	1136
spi_sync_locked .....	1137
spi_bus_lock .....	1138
spi_bus_unlock .....	1139
spi_write_then_read .....	1140
11. I <sup>2</sup> C and SMBus Subsystem .....	1141
struct i2c_driver .....	1142
struct i2c_client .....	1144
struct i2c_board_info .....	1145
I2C_BOARD_INFO .....	1146
struct i2c_algorithm .....	1147
struct i2c_bus_recovery_info .....	1148
struct i2c_adapter_quirks .....	1149
module_i2c_driver .....	1150
i2c_register_board_info .....	1151
i2c_verify_client .....	1152
i2c_lock_adapter .....	1153
i2c_unlock_adapter .....	1154
i2c_new_device .....	1155
i2c_unregister_device .....	1156
i2c_new_dummy .....	1157
i2c_verify_adapter .....	1158
i2c_add_adapter .....	1159
i2c_add_numbered_adapter .....	1160
i2c_del_adapter .....	1161
i2c_del_driver .....	1162
i2c_use_client .....	1163
i2c_release_client .....	1164
__i2c_transfer .....	1165
i2c_transfer .....	1166
i2c_master_send .....	1167
i2c_master_recv .....	1168
i2c_smbus_read_byte .....	1169
i2c_smbus_write_byte .....	1170
i2c_smbus_read_byte_data .....	1171
i2c_smbus_write_byte_data .....	1172



i2c_smbus_read_word_data .....	1173
i2c_smbus_write_word_data .....	1174
i2c_smbus_read_block_data .....	1175
i2c_smbus_write_block_data .....	1176
i2c_smbus_xfer .....	1177
i2c_smbus_read_i2c_block_data_or_emulated .....	1178
12. High Speed Synchronous Serial Interface (HSI) .....	1179
struct hsi_channel .....	1180
struct hsi_config .....	1181
struct hsi_board_info .....	1182
struct hsi_client .....	1183
struct hsi_client_driver .....	1184
struct hsi_msg .....	1185
struct hsi_port .....	1186
struct hsi_controller .....	1187
hsi_id .....	1188
hsi_port_id .....	1189
hsi_setup .....	1190
hsi_flush .....	1191
hsi_async_read .....	1192
hsi_async_write .....	1193
hsi_start_tx .....	1194
hsi_stop_tx .....	1195
hsi_port_unregister_clients .....	1196
hsi_unregister_controller .....	1197
hsi_register_controller .....	1198
hsi_register_client_driver .....	1199
hsi_put_controller .....	1200
hsi_alloc_controller .....	1201
hsi_free_msg .....	1202
hsi_alloc_msg .....	1203
hsi_async .....	1204
hsi_claim_port .....	1205
hsi_release_port .....	1206
hsi_register_port_event .....	1207
hsi_unregister_port_event .....	1208
hsi_event .....	1209
hsi_get_channel_id_by_name .....	1210
13. Pulse-Width Modulation (PWM) .....	1211
enum pwm_polarity .....	1212
struct pwm_device .....	1213
struct pwm_ops .....	1214
struct pwm_chip .....	1215
pwm_set_chip_data .....	1216
pwm_get_chip_data .....	1217
pwmchip_add_with_polarity .....	1218
pwmchip_add .....	1219
pwmchip_remove .....	1220
pwm_request .....	1221
pwm_request_from_chip .....	1222
pwm_free .....	1223
pwm_config .....	1224
pwm_set_polarity .....	1225
pwm_enable .....	1226

pwm_disable .....	1227
of_pwm_get .....	1228
pwm_get .....	1229
pwm_put .....	1230
devm_pwm_get .....	1231
devm_of_pwm_get .....	1232
devm_pwm_put .....	1233
pwm_can_sleep .....	1234

---

# **Chapter 1. Driver Basics**

## **Driver Entry and Exit points**

## Name

include/linux/init.h — Document generation inconsistency

## Oops

### Warning

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

## Atomic and pointer manipulation

## Name

`atomic_read` — read atomic variable

## Synopsis

```
int atomic_read (const atomic_t * v);
```

## Arguments

`v` pointer of type `atomic_t`

## Description

Atomically reads the value of `v`.

## Name

`atomic_set` — set atomic variable

## Synopsis

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

## Arguments

*v* pointer of type `atomic_t`

*i* required value

## Description

Atomically sets the value of *v* to *i*.

## Name

`atomic_add` — add integer to atomic variable

## Synopsis

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

## Arguments

*i* integer value to add

*v* pointer of type `atomic_t`

## Description

Atomically adds *i* to *v*.

## Name

`atomic_sub` — subtract integer from atomic variable

## Synopsis

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

## Arguments

*i* integer value to subtract

*v* pointer of type `atomic_t`

## Description

Atomically subtracts *i* from *v*.



## Name

`atomic_sub_and_test` — subtract value from variable and test result

## Synopsis

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

## Arguments

*i* integer value to subtract

*v* pointer of type `atomic_t`

## Description

Atomically subtracts *i* from *v* and returns true if the result is zero, or false for all other cases.

## Name

`atomic_inc` — increment atomic variable

## Synopsis

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

## Arguments

`v` pointer of type `atomic_t`

## Description

Atomically increments `v` by 1.

## Name

`atomic_dec` — decrement atomic variable

## Synopsis

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

## Arguments

`v` pointer of type `atomic_t`

## Description

Atomically decrements `v` by 1.

## Name

`atomic_dec_and_test` — decrement and test

## Synopsis

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

## Arguments

`v` pointer of type `atomic_t`

## Description

Atomically decrements `v` by 1 and returns true if the result is 0, or false for all other cases.

## Name

`atomic_inc_and_test` — increment and test

## Synopsis

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

## Arguments

`v` pointer of type `atomic_t`

## Description

Atomically increments `v` by 1 and returns true if the result is zero, or false for all other cases.

## Name

`atomic_add_negative` — add and test if negative

## Synopsis

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

## Arguments

*i* integer value to add

*v* pointer of type `atomic_t`

## Description

Atomically adds *i* to *v* and returns true if the result is negative, or false when result is greater than or equal to zero.

## Name

`atomic_add_return` — add integer and return

## Synopsis

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

## Arguments

*i* integer value to add

*v* pointer of type `atomic_t`

## Description

Atomically adds *i* to *v* and returns *i* + *v*

## Name

`atomic_sub_return` — subtract integer and return

## Synopsis

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

## Arguments

*i* integer value to subtract

*v* pointer of type `atomic_t`

## Description

Atomically subtracts *i* from *v* and returns *v* - *i*



## Name

`__atomic_add_unless` — add unless the number is already a given value

## Synopsis

```
int __atomic_add_unless (atomic_t * v, int a, int u);
```

## Arguments

*v* pointer of type `atomic_t`

*a* the amount to add to *v*...

*u* ...unless *v* is equal to *u*.

## Description

Atomically adds *a* to *v*, so long as *v* was not already *u*. Returns the old value of *v*.

## Name

`atomic_inc_short` — increment of a short integer

## Synopsis

```
short int atomic_inc_short (short int * v);
```

## Arguments

`v` pointer to type `int`

## Description

Atomically adds 1 to `v` Returns the new value of `u`

# Delaying, scheduling, and timer routines

## Name

struct prev\_cputime — snaphsot of system and user cputime

## Synopsis

```
struct prev_cputime {  
#ifndef CONFIG_VIRT_CPU_ACCOUNTING_NATIVE  
    cputime_t utime;  
    cputime_t stime;  
    raw_spinlock_t lock;  
#endif  
};
```

## Members

utime	time spent in user mode
stime	time spent in system mode
lock	protects the above two fields

## Description

Stores previous user/system time values such that we can guarantee monotonicity.

## Name

struct task\_cputime — collected CPU time counts

## Synopsis

```
struct task_cputime {  
    cputime_t utime;  
    cputime_t stime;  
    unsigned long long sum_exec_runtime;  
};
```

## Members

utime	time spent in user mode, in cputime_t units
stime	time spent in kernel mode, in cputime_t units
sum_exec_runtime	total time spent on the CPU, in nanoseconds

## Description

This structure groups together three kinds of CPU time that are tracked for threads and thread groups. Most things considering CPU time want to group these counts together and treat all three of them in parallel.

## Name

struct thread\_group\_cputimer — thread group interval timer counts

## Synopsis

```
struct thread_group_cputimer {  
    struct task_cputime_atomic cputime_atomic;  
    bool running;  
    bool checking_timer;  
};
```

## Members

<code>cputime_atomic</code>	atomic thread group interval timers.
<code>running</code>	true when there are timers running and <i>cputime_atomic</i> receives updates.
<code>checking_timer</code>	true when a thread in the group is in the process of checking for thread group timers.

## Description

This structure contains the version of `task_cputime`, above, that is used for thread group CPU timer calculations.

## Name

`pid_alive` — check that a task structure is not stale

## Synopsis

```
int pid_alive (const struct task_struct * p);
```

## Arguments

*p* Task structure to be checked.

## Description

Test if a process is not yet dead (at most zombie state) If `pid_alive` fails, then pointers within the task structure can be stale and must not be dereferenced.

## Return

1 if the process is alive. 0 otherwise.

## Name

`is_global_init` — check if a task structure is init. Since init is free to have sub-threads we need to check `tgid`.

## Synopsis

```
int is_global_init (struct task_struct * tsk);
```

## Arguments

*tsk* Task structure to be checked.

## Description

Check if a task structure is the first user space task the kernel created.

## Return

1 if the task structure is init. 0 otherwise.

## Name

`task_nice` — return the nice value of a given task.

## Synopsis

```
int task_nice (const struct task_struct * p);
```

## Arguments

*p* the task in question.

## Return

The nice value [ -20 ... 0 ... 19 ].



## Name

`is_idle_task` — is the specified task an idle task?

## Synopsis

```
bool is_idle_task (const struct task_struct * p);
```

## Arguments

*p* the task in question.

## Return

1 if *p* is an idle task. 0 otherwise.

## Name

`threadgroup_change_begin` — mark the beginning of changes to a threadgroup

## Synopsis

```
void threadgroup_change_begin (struct task_struct * tsk);
```

## Arguments

*tsk* task causing the changes

## Description

All operations which modify a threadgroup - a new thread joining the group, death of a member thread (the assertion of `PF_EXITING`) and `exec(2)` dethreading the process and replacing the leader - are wrapped by `threadgroup_change_{begin|end}()`. This is to provide a place which subsystems needing threadgroup stability can hook into for synchronization.

## Name

`threadgroup_change_end` — mark the end of changes to a threadgroup

## Synopsis

```
void threadgroup_change_end (struct task_struct * tsk);
```

## Arguments

*tsk* task causing the changes

## Description

See `threadgroup_change_begin`.

## Name

`wake_up_process` — Wake up a specific process

## Synopsis

```
int wake_up_process (struct task_struct * p);
```

## Arguments

*p* The process to be woken up.

## Description

Attempt to wake up the nominated process and move it to the set of runnable processes.

## Return

1 if the process was woken up, 0 if it was already running.

It may be assumed that this function implies a write memory barrier before changing the task state if and only if any tasks are woken up.

## Name

`preempt_notifier_register` — tell me when current is being preempted & rescheduled

## Synopsis

```
void preempt_notifier_register (struct preempt_notifier * notifier);
```

## Arguments

*notifier*    notifier struct to register

## Name

`preempt_notifier_unregister` — no longer interested in preemption notifications

## Synopsis

```
void preempt_notifier_unregister (struct preempt_notifier * notifier);
```

## Arguments

*notifier* notifier struct to unregister

## Description

This is *\*not\** safe to call from within a preemption notifier.

## Name

`preempt_schedule_notrace` — `preempt_schedule` called by tracing

## Synopsis

```
__visible void __sched notrace preempt_schedule_notrace ( void );
```

## Arguments

*void* no arguments

## Description

The tracing infrastructure uses `preempt_enable_notrace` to prevent recursion and tracing preempt enabling caused by the tracing infrastructure itself. But as tracing can happen in areas coming from userspace or just about to enter userspace, a preempt enable can occur before `user_exit` is called. This will cause the scheduler to be called when the system is still in usermode.

To prevent this, the `preempt_enable_notrace` will use this function instead of `preempt_schedule` to exit user context if needed before calling the scheduler.

## Name

`sched_setscheduler` — change the scheduling policy and/or RT priority of a thread.

## Synopsis

```
int sched_setscheduler (struct task_struct * p, int policy, const struct  
sched_param * param);
```

## Arguments

*p*            the task in question.

*policy*    new policy.

*param*    structure containing the new RT priority.

## Return

0 on success. An error code otherwise.

NOTE that the task may be already dead.



## Name

`sched_setscheduler_nocheck` — change the scheduling policy and/or RT priority of a thread from kernelspace.

## Synopsis

```
int sched_setscheduler_nocheck (struct task_struct * p, int policy,  
const struct sched_param * param);
```

## Arguments

*p*            the task in question.

*policy*      new policy.

*param*      structure containing the new RT priority.

## Description

Just like `sched_setscheduler`, only don't bother checking if the current context has permission. For example, this is needed in `stop_machine`: we create temporary high priority worker threads, but our caller might not have that capability.

## Return

0 on success. An error code otherwise.

## Name

`yield` — yield the current processor to other threads.

## Synopsis

```
void __sched yield ( void );
```

## Arguments

*void* no arguments

## Description

Do not ever use this function, there's a 99% chance you're doing it wrong.

The scheduler is at all times free to pick the calling task as the most eligible task to run, if removing the `yield` call from your code breaks it, its already broken.

## Typical broken usage is

```
while (!event) yield;
```

where one assumes that `yield` will let 'the other' process run that will make event true. If the current task is a `SCHED_FIFO` task that will never happen. Never use `yield` as a progress guarantee!!

If you want to use `yield` to wait for something, use `wait_event`. If you want to use `yield` to be 'nice' for others, use `cond_resched`. If you still want to use `yield`, do not!

## Name

`yield_to` — yield the current processor to another thread in your thread group, or accelerate that thread toward the processor it's on.

## Synopsis

```
int __sched yield_to (struct task_struct * p, bool preempt);
```

## Arguments

*p*            target task

*preempt*    whether task preemption is allowed or not

## Description

It's the caller's job to ensure that the target task struct can't go away on us before we can do any checks.

## Return

true (>0) if we indeed boosted the target task. false (0) if we failed to boost the target. -ESRCH if there's no task to yield to.

## Name

`cpupri_find` — find the best (lowest-pri) CPU in the system

## Synopsis

```
int cpupri_find (struct cpupri * cp, struct task_struct * p, struct
cpumask * lowest_mask);
```

## Arguments

<i>cp</i>	The cpupri context
<i>p</i>	The task
<i>lowest_mask</i>	A mask to fill in with selected CPUs (or NULL)

## Note

This function returns the recommended CPUs as calculated during the current invocation. By the time the call returns, the CPUs may have in fact changed priorities any number of times. While not ideal, it is not an issue of correctness since the normal rebalancer logic will correct any discrepancies created by racing against the uncertainty of the current priority configuration.

## Return

(int)bool - CPUs were found

## Name

`cpupri_set` — update the cpu priority setting

## Synopsis

```
void cpupri_set (struct cpupri * cp, int cpu, int newpri);
```

## Arguments

*cp*            The cpupri context

*cpu*           The target cpu

*newpri*       The priority (INVALID-RT99) to assign to this CPU

## Note

Assumes `cpu_rq(cpu)->lock` is locked

## Returns

(void)

## Name

`cpupri_init` — initialize the `cpupri` structure

## Synopsis

```
int cpupri_init (struct cpupri * cp);
```

## Arguments

*cp*    The `cpupri` context

## Return

-ENOMEM on memory allocation failure.

## Name

`cpupri_cleanup` — clean up the `cpupri` structure

## Synopsis

```
void cpupri_cleanup (struct cpupri * cp);
```

## Arguments

*cp* The `cpupri` context

## Name

`get_sd_load_idx` — Obtain the load index for a given sched domain.

## Synopsis

```
int get_sd_load_idx (struct sched_domain * sd, enum cpu_idle_type idle);
```

## Arguments

*sd*      The sched\_domain whose load\_idx is to be obtained.

*idle*    The idle status of the CPU for whose sd load\_idx is obtained.

## Return

The load index.



## Name

`update_sg_lb_stats` — Update `sched_group`'s statistics for load balancing.

## Synopsis

```
void update_sg_lb_stats (struct lb_env * env, struct sched_group *  
group, int load_idx, int local_group, struct sg_lb_stats * sgs, bool  
* overload);
```

## Arguments

<i>env</i>	The load balancing environment.
<i>group</i>	<code>sched_group</code> whose statistics are to be updated.
<i>load_idx</i>	Load index of <code>sched_domain</code> of <code>this_cpu</code> for load calc.
<i>local_group</i>	Does group contain <code>this_cpu</code> .
<i>sgs</i>	variable to hold the statistics for this group.
<i>overload</i>	Indicate more than one runnable task for any CPU.

## Name

`update_sd_pick_busiest` — return 1 on busiest group

## Synopsis

```
bool update_sd_pick_busiest (struct lb_env * env, struct sd_lb_stats *  
sds, struct sched_group * sg, struct sg_lb_stats * sgs);
```

## Arguments

*env* The load balancing environment.

*sds* sched\_domain statistics

*sg* sched\_group candidate to be checked for being the busiest

*sgs* sched\_group statistics

## Description

Determine if *sg* is a busier group than the previously selected busiest group.

## Return

true if *sg* is a busier group than the previously selected busiest group. false otherwise.

## Name

`update_sd_lb_stats` — Update sched\_domain's statistics for load balancing.

## Synopsis

```
void update_sd_lb_stats (struct lb_env * env, struct sd_lb_stats * sds);
```

## Arguments

*env*    The load balancing environment.

*sds*    variable to hold the statistics for this sched\_domain.

## Name

`check_asym_packing` — Check to see if the group is packed into the sched domain.

## Synopsis

```
int check_asym_packing (struct lb_env * env, struct sd_lb_stats * sds);
```

## Arguments

*env* The load balancing environment.

*sds* Statistics of the sched\_domain which is to be packed

## Description

This is primarily intended to be used at the sibling level. Some cores like POWER7 prefer to use lower numbered SMT threads. In the case of POWER7, it can move to lower SMT modes only when higher threads are idle. When in lower SMT modes, the threads will perform better since they share less core resources. Hence when we have idle threads, we want them to be the higher ones.

This packing function is run on idle threads. It checks to see if the busiest CPU in this domain (core in the P7 case) has a higher CPU number than the packing function is being run on. Here we are assuming lower CPU number will be equivalent to lower a SMT thread number.

## Return

1 when packing is required and a task should be moved to this CPU. The amount of the imbalance is returned in *\*imbalance*.

## Name

`fix_small_imbalance` — Calculate the minor imbalance that exists amongst the groups of a `sched_domain`, during load balancing.

## Synopsis

```
void fix_small_imbalance (struct lb_env * env, struct sd_lb_stats *  
sds);
```

## Arguments

*env* The load balancing environment.

*sds* Statistics of the `sched_domain` whose imbalance is to be calculated.

## Name

`calculate_imbalance` — Calculate the amount of imbalance present within the groups of a given `sched_domain` during load balance.

## Synopsis

```
void calculate_imbalance (struct lb_env * env, struct sd_lb_stats *  
sds);
```

## Arguments

*env* load balance environment

*sds* statistics of the `sched_domain` whose imbalance is to be calculated.

## Name

`find_busiest_group` — Returns the busiest group within the `sched_domain` if there is an imbalance. If there isn't an imbalance, and the user has opted for power-savings, it returns a group whose CPUs can be put to idle by rebalancing those tasks elsewhere, if such a group exists.

## Synopsis

```
struct sched_group * find_busiest_group (struct lb_env * env);
```

## Arguments

*env* The load balancing environment.

## Description

Also calculates the amount of weighted load which should be moved to restore balance.

## Return

- The busiest group if imbalance exists. - If no imbalance and user has opted for power-savings balance, return the least loaded group whose CPUs can be put to idle by rebalancing its tasks onto our group.

## Name

DECLARE\_COMPLETION — declare and initialize a completion structure

## Synopsis

```
DECLARE_COMPLETION ( work );
```

## Arguments

*work* identifier for the completion structure

## Description

This macro declares and initializes a completion structure. Generally used for static declarations. You should use the `_ONSTACK` variant for automatic variables.



## Name

`DECLARE_COMPLETION_ONSTACK` — declare and initialize a completion structure

## Synopsis

```
DECLARE_COMPLETION_ONSTACK ( work );
```

## Arguments

*work* identifier for the completion structure

## Description

This macro declares and initializes a completion structure on the kernel stack.

## Name

`init_completion` — Initialize a dynamically allocated completion

## Synopsis

```
void init_completion (struct completion * x);
```

## Arguments

*x* pointer to completion structure that is to be initialized

## Description

This inline function will initialize a dynamically created completion structure.

## Name

reinit\_completion — reinitialize a completion structure

## Synopsis

```
void reinit_completion (struct completion * x);
```

## Arguments

*x* pointer to completion structure that is to be reinitialized

## Description

This inline function should be used to reinitialize a completion structure so it can be reused. This is especially important after `complete_all` is used.

## Name

`__round_jiffies` — function to round jiffies to a full second

## Synopsis

```
unsigned long __round_jiffies (unsigned long j, int cpu);
```

## Arguments

*j*      the time in (absolute) jiffies that should be rounded

*cpu*    the processor number on which the timeout will happen

## Description

`__round_jiffies` rounds an absolute time in the future (in jiffies) up or down to (approximately) full seconds. This is useful for timers for which the exact time they fire does not matter too much, as long as they fire approximately every X seconds.

By rounding these timers to whole seconds, all such timers will fire at the same time, rather than at various times spread out. The goal of this is to have the CPU wake up less, which saves power.

The exact rounding is skewed for each processor to avoid all processors firing at the exact same time, which could lead to lock contention or spurious cache line bouncing.

The return value is the rounded version of the *j* parameter.

## Name

`__round_jiffies_relative` — function to round jiffies to a full second

## Synopsis

```
unsigned long __round_jiffies_relative (unsigned long j, int cpu);
```

## Arguments

*j*      the time in (relative) jiffies that should be rounded

*cpu*    the processor number on which the timeout will happen

## Description

`__round_jiffies_relative` rounds a time delta in the future (in jiffies) up or down to (approximately) full seconds. This is useful for timers for which the exact time they fire does not matter too much, as long as they fire approximately every X seconds.

By rounding these timers to whole seconds, all such timers will fire at the same time, rather than at various times spread out. The goal of this is to have the CPU wake up less, which saves power.

The exact rounding is skewed for each processor to avoid all processors firing at the exact same time, which could lead to lock contention or spurious cache line bouncing.

The return value is the rounded version of the *j* parameter.

## Name

`round_jiffies` — function to round jiffies to a full second

## Synopsis

```
unsigned long round_jiffies (unsigned long j);
```

## Arguments

*j* the time in (absolute) jiffies that should be rounded

## Description

`round_jiffies` rounds an absolute time in the future (in jiffies) up or down to (approximately) full seconds. This is useful for timers for which the exact time they fire does not matter too much, as long as they fire approximately every X seconds.

By rounding these timers to whole seconds, all such timers will fire at the same time, rather than at various times spread out. The goal of this is to have the CPU wake up less, which saves power.

The return value is the rounded version of the *j* parameter.

## Name

`round_jiffies_relative` — function to round jiffies to a full second

## Synopsis

```
unsigned long round_jiffies_relative (unsigned long j);
```

## Arguments

*j* the time in (relative) jiffies that should be rounded

## Description

`round_jiffies_relative` rounds a time delta in the future (in jiffies) up or down to (approximately) full seconds. This is useful for timers for which the exact time they fire does not matter too much, as long as they fire approximately every X seconds.

By rounding these timers to whole seconds, all such timers will fire at the same time, rather than at various times spread out. The goal of this is to have the CPU wake up less, which saves power.

The return value is the rounded version of the *j* parameter.

## Name

`__round_jiffies_up` — function to round jiffies up to a full second

## Synopsis

```
unsigned long __round_jiffies_up (unsigned long j, int cpu);
```

## Arguments

*j*      the time in (absolute) jiffies that should be rounded

*cpu*    the processor number on which the timeout will happen

## Description

This is the same as `__round_jiffies` except that it will never round down. This is useful for timeouts for which the exact time of firing does not matter too much, as long as they don't fire too early.



## Name

`__round_jiffies_up_relative` — function to round jiffies up to a full second

## Synopsis

```
unsigned long __round_jiffies_up_relative (unsigned long j, int cpu);
```

## Arguments

*j*      the time in (relative) jiffies that should be rounded

*cpu*    the processor number on which the timeout will happen

## Description

This is the same as `__round_jiffies_relative` except that it will never round down. This is useful for timeouts for which the exact time of firing does not matter too much, as long as they don't fire too early.

## Name

`round_jiffies_up` — function to round jiffies up to a full second

## Synopsis

```
unsigned long round_jiffies_up (unsigned long j);
```

## Arguments

*j* the time in (absolute) jiffies that should be rounded

## Description

This is the same as `round_jiffies` except that it will never round down. This is useful for timeouts for which the exact time of firing does not matter too much, as long as they don't fire too early.

## Name

`round_jiffies_up_relative` — function to round jiffies up to a full second

## Synopsis

```
unsigned long round_jiffies_up_relative (unsigned long j);
```

## Arguments

*j* the time in (relative) jiffies that should be rounded

## Description

This is the same as `round_jiffies_relative` except that it will never round down. This is useful for timeouts for which the exact time of firing does not matter too much, as long as they don't fire too early.

## Name

`set_timer_slack` — set the allowed slack for a timer

## Synopsis

```
void set_timer_slack (struct timer_list * timer, int slack_hz);
```

## Arguments

*timer*        the timer to be modified

*slack\_hz*    the amount of time (in jiffies) allowed for rounding

## Description

Set the amount of time, in jiffies, that a certain timer has in terms of slack. By setting this value, the timer subsystem will schedule the actual timer somewhere between the time `mod_timer` asks for, and that time plus the slack.

By setting the slack to -1, a percentage of the delay is used instead.

## Name

`init_timer_key` — initialize a timer

## Synopsis

```
void init_timer_key (struct timer_list * timer, unsigned int flags,  
const char * name, struct lock_class_key * key);
```

## Arguments

*timer*    the timer to be initialized

*flags*    timer flags

*name*     name of the timer

*key*      lockdep class key of the fake lock used for tracking timer sync lock dependencies

## Description

`init_timer_key` must be done to a timer prior calling *any* of the other timer functions.

## Name

`mod_timer_pending` — modify a pending timer's timeout

## Synopsis

```
int mod_timer_pending (struct timer_list * timer, unsigned long
expires);
```

## Arguments

*timer*      the pending timer to be modified

*expires*    new timeout in jiffies

## Description

`mod_timer_pending` is the same for pending timers as `mod_timer`, but will not re-activate and modify already deleted timers.

It is useful for unserialized use of timers.

## Name

`mod_timer` — modify a timer's timeout

## Synopsis

```
int mod_timer (struct timer_list * timer, unsigned long expires);
```

## Arguments

*timer*      the timer to be modified

*expires*    new timeout in jiffies

## Description

`mod_timer` is a more efficient way to update the `expire` field of an active timer (if the timer is inactive it will be activated)

`mod_timer(timer, expires)` is equivalent to:

```
del_timer(timer); timer->expires = expires; add_timer(timer);
```

Note that if there are multiple unserialized concurrent users of the same timer, then `mod_timer` is the only safe way to modify the timeout, since `add_timer` cannot modify an already running timer.

The function returns whether it has modified a pending timer or not. (ie. `mod_timer` of an inactive timer returns 0, `mod_timer` of an active timer returns 1.)

## Name

`mod_timer_pinned` — modify a timer's timeout

## Synopsis

```
int mod_timer_pinned (struct timer_list * timer, unsigned long expires);
```

## Arguments

*timer*      the timer to be modified

*expires*    new timeout in jiffies

## Description

`mod_timer_pinned` is a way to update the `expire` field of an active timer (if the timer is inactive it will be activated) and to ensure that the timer is scheduled on the current CPU.

Note that this does not prevent the timer from being migrated when the current CPU goes offline. If this is a problem for you, use CPU-hotplug notifiers to handle it correctly, for example, cancelling the timer when the corresponding CPU goes offline.

`mod_timer_pinned(timer, expires)` is equivalent to:

```
del_timer(timer); timer->expires = expires; add_timer(timer);
```



## Name

`add_timer` — start a timer

## Synopsis

```
void add_timer (struct timer_list * timer);
```

## Arguments

*timer* the timer to be added

## Description

The kernel will do a `->function(->data)` callback from the timer interrupt at the `->expires` point in the future. The current time is 'jiffies'.

The timer's `->expires`, `->function` (and if the handler uses it, `->data`) fields must be set prior calling this function.

Timers with an `->expires` field in the past will be executed in the next timer tick.

## Name

`add_timer_on` — start a timer on a particular CPU

## Synopsis

```
void add_timer_on (struct timer_list * timer, int cpu);
```

## Arguments

*timer*    the timer to be added

*cpu*      the CPU to start it on

## Description

This is not very scalable on SMP. Double adds are not possible.

## Name

`del_timer` — deactivate a timer.

## Synopsis

```
int del_timer (struct timer_list * timer);
```

## Arguments

*timer* the timer to be deactivated

## Description

`del_timer` deactivates a timer - this works on both active and inactive timers.

The function returns whether it has deactivated a pending timer or not. (ie. `del_timer` of an inactive timer returns 0, `del_timer` of an active timer returns 1.)

## Name

`try_to_del_timer_sync` — Try to deactivate a timer

## Synopsis

```
int try_to_del_timer_sync (struct timer_list * timer);
```

## Arguments

*timer*   timer to del

## Description

This function tries to deactivate a timer. Upon successful (ret  $\geq$  0) exit the timer is not queued and the handler is not running on any CPU.

## Name

`del_timer_sync` — deactivate a timer and wait for the handler to finish.

## Synopsis

```
int del_timer_sync (struct timer_list * timer);
```

## Arguments

*timer* the timer to be deactivated

## Description

This function only differs from `del_timer` on SMP: besides deactivating the timer it also makes sure the handler has finished executing on other CPUs.

## Synchronization rules

Callers must prevent restarting of the timer, otherwise this function is meaningless. It must not be called from interrupt contexts unless the timer is an irqsafe one. The caller must not hold locks which would prevent completion of the timer's handler. The timer's handler must not call `add_timer_on`. Upon exit the timer is not queued and the handler is not running on any CPU.

## Note

For !irqsafe timers, you must not hold locks that are held in interrupt context while calling this function. Even if the lock has nothing to do with the timer in question. Here's why:

```
CPU0  CPU1  ----  ----  <SOFTIRQ>  call_timer_fn;  base->running_timer = mytimer;
spin_lock_irq(somelock);  <IRQ>  spin_lock(somelock);  del_timer_sync(mytimer);  while (base-
->running_timer == mytimer);
```

Now `del_timer_sync` will never return and never release `somelock`. The interrupt on the other CPU is waiting to grab `somelock` but it has interrupted the softirq that CPU0 is waiting to finish.

The function returns whether it has deactivated a pending timer or not.

## Name

`schedule_timeout` — sleep until timeout

## Synopsis

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

## Arguments

*timeout* timeout value in jiffies

## Description

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

You can set the task state as follows -

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

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

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

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

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

## Name

`msleep` — sleep safely even with waitqueue interruptions

## Synopsis

```
void msleep (unsigned int msecs);
```

## Arguments

*msecs*    Time in milliseconds to sleep for

## Name

`msleep_interruptible` — sleep waiting for signals

## Synopsis

```
unsigned long msleep_interruptible (unsigned int msecs);
```

## Arguments

*msecs*    Time in milliseconds to sleep for



## Name

`usleep_range` — Drop in replacement for `udelay` where wakeup is flexible

## Synopsis

```
void __sched usleep_range (unsigned long min, unsigned long max);
```

## Arguments

*min* Minimum time in usecs to sleep

*max* Maximum time in usecs to sleep

## Wait queues and Wake events

## Name

`wait_event` — sleep until a condition gets true

## Synopsis

```
wait_event ( wq, condition);
```

## Arguments

*wq*                    the waitqueue to wait on

*condition*    a C expression for the event to wait for

## Description

The process is put to sleep (`TASK_UNINTERRUPTIBLE`) until the *condition* evaluates to true. The *condition* is checked each time the waitqueue *wq* is woken up.

`wake_up` has to be called after changing any variable that could change the result of the wait condition.

## Name

`wait_event_freezable` — sleep (or freeze) until a condition gets true

## Synopsis

```
wait_event_freezable ( wq, condition );
```

## Arguments

*wq*                    the waitqueue to wait on

*condition*    a C expression for the event to wait for

## Description

The process is put to sleep (`TASK_INTERRUPTIBLE` -- so as not to contribute to system load) until the *condition* evaluates to true. The *condition* is checked each time the waitqueue *wq* is woken up.

`wake_up` has to be called after changing any variable that could change the result of the wait condition.

## Name

`wait_event_timeout` — sleep until a condition gets true or a timeout elapses

## Synopsis

```
wait_event_timeout ( wq, condition, timeout );
```

## Arguments

<i>wq</i>	the waitqueue to wait on
<i>condition</i>	a C expression for the event to wait for
<i>timeout</i>	timeout, in jiffies

## Description

The process is put to sleep (`TASK_UNINTERRUPTIBLE`) until the *condition* evaluates to true. The *condition* is checked each time the waitqueue *wq* is woken up.

`wake_up` has to be called after changing any variable that could change the result of the wait condition.

## Returns

0 if the *condition* evaluated to false after the *timeout* elapsed, 1 if the *condition* evaluated to true after the *timeout* elapsed, or the remaining jiffies (at least 1) if the *condition* evaluated to true before the *timeout* elapsed.

## Name

`wait_event_cmd` — sleep until a condition gets true

## Synopsis

```
wait_event_cmd ( wq, condition, cmd1, cmd2 );
```

## Arguments

<i>wq</i>	the waitqueue to wait on
<i>condition</i>	a C expression for the event to wait for
<i>cmd1</i>	the command will be executed before sleep
<i>cmd2</i>	the command will be executed after sleep

## Description

The process is put to sleep (TASK\_UNINTERRUPTIBLE) until the *condition* evaluates to true. The *condition* is checked each time the waitqueue *wq* is woken up.

`wake_up` has to be called after changing any variable that could change the result of the wait condition.

## Name

`wait_event_interruptible` — sleep until a condition gets true

## Synopsis

```
wait_event_interruptible ( wq, condition);
```

## Arguments

*wq*                    the waitqueue to wait on

*condition*    a C expression for the event to wait for

## Description

The process is put to sleep (TASK\_INTERRUPTIBLE) until the *condition* evaluates to true or a signal is received. The *condition* is checked each time the waitqueue *wq* is woken up.

`wake_up` has to be called after changing any variable that could change the result of the wait condition.

The function will return `-ERESTARTSYS` if it was interrupted by a signal and 0 if *condition* evaluated to true.

## Name

`wait_event_interruptible_timeout` — sleep until a condition gets true or a timeout elapses

## Synopsis

```
wait_event_interruptible_timeout ( wq, condition, timeout);
```

## Arguments

<i>wq</i>	the waitqueue to wait on
<i>condition</i>	a C expression for the event to wait for
<i>timeout</i>	timeout, in jiffies

## Description

The process is put to sleep (`TASK_INTERRUPTIBLE`) until the *condition* evaluates to true or a signal is received. The *condition* is checked each time the waitqueue *wq* is woken up.

`wake_up` has to be called after changing any variable that could change the result of the wait condition.

## Returns

0 if the *condition* evaluated to false after the *timeout* elapsed, 1 if the *condition* evaluated to true after the *timeout* elapsed, the remaining jiffies (at least 1) if the *condition* evaluated to true before the *timeout* elapsed, or `-ERESTARTSYS` if it was interrupted by a signal.

## Name

`wait_event_hrtimeout` — sleep until a condition gets true or a timeout elapses

## Synopsis

```
wait_event_hrtimeout ( wq, condition, timeout );
```

## Arguments

<i>wq</i>	the waitqueue to wait on
<i>condition</i>	a C expression for the event to wait for
<i>timeout</i>	timeout, as a <code>ktime_t</code>

## Description

The process is put to sleep (`TASK_UNINTERRUPTIBLE`) until the *condition* evaluates to true or a signal is received. The *condition* is checked each time the waitqueue *wq* is woken up.

`wake_up` has to be called after changing any variable that could change the result of the wait condition.

The function returns 0 if *condition* became true, or `-ETIME` if the timeout elapsed.



## Name

`wait_event_interruptible_hrtimeout` — sleep until a condition gets true or a timeout elapses

## Synopsis

```
wait_event_interruptible_hrtimeout ( wq, condition, timeout);
```

## Arguments

<i>wq</i>	the waitqueue to wait on
<i>condition</i>	a C expression for the event to wait for
<i>timeout</i>	timeout, as a <code>ktime_t</code>

## Description

The process is put to sleep (`TASK_INTERRUPTIBLE`) until the *condition* evaluates to true or a signal is received. The *condition* is checked each time the waitqueue *wq* is woken up.

`wake_up` has to be called after changing any variable that could change the result of the wait condition.

The function returns 0 if *condition* became true, `-ERESTARTSYS` if it was interrupted by a signal, or `-ETIME` if the timeout elapsed.

## Name

`wait_event_interruptible_locked` — sleep until a condition gets true

## Synopsis

```
wait_event_interruptible_locked ( wq, condition);
```

## Arguments

*wq*                    the waitqueue to wait on

*condition*    a C expression for the event to wait for

## Description

The process is put to sleep (TASK\_INTERRUPTIBLE) until the *condition* evaluates to true or a signal is received. The *condition* is checked each time the waitqueue *wq* is woken up.

It must be called with *wq.lock* being held. This spinlock is unlocked while sleeping but *condition* testing is done while lock is held and when this macro exits the lock is held.

The lock is locked/unlocked using `spin_lock/spin_unlock` functions which must match the way they are locked/unlocked outside of this macro.

`wake_up_locked` has to be called after changing any variable that could change the result of the wait condition.

The function will return `-ERESTARTSYS` if it was interrupted by a signal and 0 if *condition* evaluated to true.

## Name

`wait_event_interruptible_locked_irq` — sleep until a condition gets true

## Synopsis

```
wait_event_interruptible_locked_irq ( wq, condition);
```

## Arguments

*wq*                    the waitqueue to wait on

*condition*    a C expression for the event to wait for

## Description

The process is put to sleep (TASK\_INTERRUPTIBLE) until the *condition* evaluates to true or a signal is received. The *condition* is checked each time the waitqueue *wq* is woken up.

It must be called with *wq.lock* being held. This spinlock is unlocked while sleeping but *condition* testing is done while lock is held and when this macro exits the lock is held.

The lock is locked/unlocked using `spin_lock_irq/spin_unlock_irq` functions which must match the way they are locked/unlocked outside of this macro.

`wake_up_locked` has to be called after changing any variable that could change the result of the wait condition.

The function will return `-ERESTARTSYS` if it was interrupted by a signal and 0 if *condition* evaluated to true.

## Name

`wait_event_interruptible_exclusive_locked` — sleep exclusively until a condition gets true

## Synopsis

```
wait_event_interruptible_exclusive_locked ( wq, condition);
```

## Arguments

*wq*                    the waitqueue to wait on

*condition*    a C expression for the event to wait for

## Description

The process is put to sleep (TASK\_INTERRUPTIBLE) until the *condition* evaluates to true or a signal is received. The *condition* is checked each time the waitqueue *wq* is woken up.

It must be called with *wq.lock* being held. This spinlock is unlocked while sleeping but *condition* testing is done while lock is held and when this macro exits the lock is held.

The lock is locked/unlocked using `spin_lock/spin_unlock` functions which must match the way they are locked/unlocked outside of this macro.

The process is put on the wait queue with an `WQ_FLAG_EXCLUSIVE` flag set thus when other process waits process on the list if this process is awoken further processes are not considered.

`wake_up_locked` has to be called after changing any variable that could change the result of the wait condition.

The function will return `-ERESTARTSYS` if it was interrupted by a signal and 0 if *condition* evaluated to true.

## Name

`wait_event_interruptible_exclusive_locked_irq` — sleep until a condition gets true

## Synopsis

```
wait_event_interruptible_exclusive_locked_irq ( wq, condition);
```

## Arguments

*wq*                    the waitqueue to wait on

*condition*    a C expression for the event to wait for

## Description

The process is put to sleep (TASK\_INTERRUPTIBLE) until the *condition* evaluates to true or a signal is received. The *condition* is checked each time the waitqueue *wq* is woken up.

It must be called with *wq.lock* being held. This spinlock is unlocked while sleeping but *condition* testing is done while lock is held and when this macro exits the lock is held.

The lock is locked/unlocked using `spin_lock_irq/spin_unlock_irq` functions which must match the way they are locked/unlocked outside of this macro.

The process is put on the wait queue with an `WQ_FLAG_EXCLUSIVE` flag set thus when other process waits process on the list if this process is awoken further processes are not considered.

`wake_up_locked` has to be called after changing any variable that could change the result of the wait condition.

The function will return `-ERESTARTSYS` if it was interrupted by a signal and 0 if *condition* evaluated to true.

## Name

`wait_event_killable` — sleep until a condition gets true

## Synopsis

```
wait_event_killable ( wq, condition );
```

## Arguments

*wq*                      the waitqueue to wait on

*condition*    a C expression for the event to wait for

## Description

The process is put to sleep (TASK\_KILLABLE) until the *condition* evaluates to true or a signal is received. The *condition* is checked each time the waitqueue *wq* is woken up.

`wake_up` has to be called after changing any variable that could change the result of the wait condition.

The function will return `-ERESTARTSYS` if it was interrupted by a signal and 0 if *condition* evaluated to true.

## Name

`wait_event_lock_irq_cmd` — sleep until a condition gets true. The condition is checked under the lock. This is expected to be called with the lock taken.

## Synopsis

```
wait_event_lock_irq_cmd ( wq, condition, lock, cmd);
```

## Arguments

<i>wq</i>	the waitqueue to wait on
<i>condition</i>	a C expression for the event to wait for
<i>lock</i>	a locked <code>spinlock_t</code> , which will be released before <code>cmd</code> and <code>schedule</code> and reacquired afterwards.
<i>cmd</i>	a command which is invoked outside the critical section before sleep

## Description

The process is put to sleep (`TASK_UNINTERRUPTIBLE`) until the *condition* evaluates to true. The *condition* is checked each time the waitqueue *wq* is woken up.

`wake_up` has to be called after changing any variable that could change the result of the wait condition.

This is supposed to be called while holding the lock. The lock is dropped before invoking the `cmd` and going to sleep and is reacquired afterwards.

## Name

`wait_event_lock_irq` — sleep until a condition gets true. The condition is checked under the lock. This is expected to be called with the lock taken.

## Synopsis

```
wait_event_lock_irq ( wq, condition, lock);
```

## Arguments

*wq*                    the waitqueue to wait on

*condition*    a C expression for the event to wait for

*lock*                a locked `spinlock_t`, which will be released before `schedule` and reacquired afterwards.

## Description

The process is put to sleep (`TASK_UNINTERRUPTIBLE`) until the *condition* evaluates to true. The *condition* is checked each time the waitqueue *wq* is woken up.

`wake_up` has to be called after changing any variable that could change the result of the wait condition.

This is supposed to be called while holding the lock. The lock is dropped before going to sleep and is reacquired afterwards.



## Name

`wait_event_interruptible_lock_irq_cmd` — sleep until a condition gets true. The condition is checked under the lock. This is expected to be called with the lock taken.

## Synopsis

```
wait_event_interruptible_lock_irq_cmd ( wq, condition, lock, cmd);
```

## Arguments

<i>wq</i>	the waitqueue to wait on
<i>condition</i>	a C expression for the event to wait for
<i>lock</i>	a locked <code>spinlock_t</code> , which will be released before <code>cmd</code> and <code>schedule</code> and reacquired afterwards.
<i>cmd</i>	a command which is invoked outside the critical section before sleep

## Description

The process is put to sleep (`TASK_INTERRUPTIBLE`) until the *condition* evaluates to true or a signal is received. The *condition* is checked each time the waitqueue *wq* is woken up.

`wake_up` has to be called after changing any variable that could change the result of the wait condition.

This is supposed to be called while holding the lock. The lock is dropped before invoking the `cmd` and going to sleep and is reacquired afterwards.

The macro will return `-ERESTARTSYS` if it was interrupted by a signal and 0 if *condition* evaluated to true.

## Name

`wait_event_interruptible_lock_irq` — sleep until a condition gets true. The condition is checked under the lock. This is expected to be called with the lock taken.

## Synopsis

```
wait_event_interruptible_lock_irq ( wq, condition, lock);
```

## Arguments

*wq*                    the waitqueue to wait on

*condition*        a C expression for the event to wait for

*lock*                a locked `spinlock_t`, which will be released before `schedule` and reacquired afterwards.

## Description

The process is put to sleep (`TASK_INTERRUPTIBLE`) until the *condition* evaluates to true or signal is received. The *condition* is checked each time the waitqueue *wq* is woken up.

`wake_up` has to be called after changing any variable that could change the result of the wait condition.

This is supposed to be called while holding the lock. The lock is dropped before going to sleep and is reacquired afterwards.

The macro will return `-ERESTARTSYS` if it was interrupted by a signal and 0 if *condition* evaluated to true.

## Name

`wait_event_interruptible_lock_irq_timeout` — sleep until a condition gets true or a timeout elapses. The condition is checked under the lock. This is expected to be called with the lock taken.

## Synopsis

```
wait_event_interruptible_lock_irq_timeout ( wq, condition, lock,
timeout );
```

## Arguments

<i>wq</i>	the waitqueue to wait on
<i>condition</i>	a C expression for the event to wait for
<i>lock</i>	a locked <code>spinlock_t</code> , which will be released before <code>schedule</code> and reacquired afterwards.
<i>timeout</i>	timeout, in jiffies

## Description

The process is put to sleep (`TASK_INTERRUPTIBLE`) until the *condition* evaluates to true or signal is received. The *condition* is checked each time the waitqueue *wq* is woken up.

`wake_up` has to be called after changing any variable that could change the result of the wait condition.

This is supposed to be called while holding the lock. The lock is dropped before going to sleep and is reacquired afterwards.

The function returns 0 if the *timeout* elapsed, `-ERESTARTSYS` if it was interrupted by a signal, and the remaining jiffies otherwise if the condition evaluated to true before the timeout elapsed.

## Name

`wait_on_bit` — wait for a bit to be cleared

## Synopsis

```
int wait_on_bit (unsigned long * word, int bit, unsigned mode);
```

## Arguments

*word* the word being waited on, a kernel virtual address

*bit* the bit of the word being waited on

*mode* the task state to sleep in

## Description

There is a standard hashed waitqueue table for generic use. This is the part of the hashtable's accessor API that waits on a bit. For instance, if one were to have waiters on a bitflag, one would call `wait_on_bit` in threads waiting for the bit to clear. One uses `wait_on_bit` where one is waiting for the bit to clear, but has no intention of setting it. Returned value will be zero if the bit was cleared, or non-zero if the process received a signal and the mode permitted wakeup on that signal.

## Name

`wait_on_bit_io` — wait for a bit to be cleared

## Synopsis

```
int wait_on_bit_io (unsigned long * word, int bit, unsigned mode);
```

## Arguments

*word* the word being waited on, a kernel virtual address

*bit* the bit of the word being waited on

*mode* the task state to sleep in

## Description

Use the standard hashed waitqueue table to wait for a bit to be cleared. This is similar to `wait_on_bit`, but calls `io_schedule` instead of `schedule` for the actual waiting.

Returned value will be zero if the bit was cleared, or non-zero if the process received a signal and the mode permitted wakeup on that signal.

## Name

`wait_on_bit_timeout` — wait for a bit to be cleared or a timeout elapses

## Synopsis

```
int wait_on_bit_timeout (unsigned long * word, int bit, unsigned mode,  
unsigned long timeout);
```

## Arguments

*word*        the word being waited on, a kernel virtual address

*bit*        the bit of the word being waited on

*mode*       the task state to sleep in

*timeout*    timeout, in jiffies

## Description

Use the standard hashed waitqueue table to wait for a bit to be cleared. This is similar to `wait_on_bit`, except also takes a timeout parameter.

Returned value will be zero if the bit was cleared before the *timeout* elapsed, or non-zero if the *timeout* elapsed or process received a signal and the mode permitted wakeup on that signal.

## Name

`wait_on_bit_action` — wait for a bit to be cleared

## Synopsis

```
int wait_on_bit_action (unsigned long * word, int bit, wait_bit_action_f  
* action, unsigned mode);
```

## Arguments

*word*      the word being waited on, a kernel virtual address

*bit*        the bit of the word being waited on

*action*    the function used to sleep, which may take special actions

*mode*      the task state to sleep in

## Description

Use the standard hashed waitqueue table to wait for a bit to be cleared, and allow the waiting action to be specified. This is like `wait_on_bit` but allows fine control of how the waiting is done.

Returned value will be zero if the bit was cleared, or non-zero if the process received a signal and the mode permitted wakeup on that signal.

## Name

`wait_on_bit_lock` — wait for a bit to be cleared, when wanting to set it

## Synopsis

```
int wait_on_bit_lock (unsigned long * word, int bit, unsigned mode);
```

## Arguments

*word* the word being waited on, a kernel virtual address

*bit* the bit of the word being waited on

*mode* the task state to sleep in

## Description

There is a standard hashed waitqueue table for generic use. This is the part of the hashtable's accessor API that waits on a bit when one intends to set it, for instance, trying to lock bitflags. For instance, if one were to have waiters trying to set bitflag and waiting for it to clear before setting it, one would call `wait_on_bit` in threads waiting to be able to set the bit. One uses `wait_on_bit_lock` where one is waiting for the bit to clear with the intention of setting it, and when done, clearing it.

Returns zero if the bit was (eventually) found to be clear and was set. Returns non-zero if a signal was delivered to the process and the *mode* allows that signal to wake the process.



## Name

`wait_on_bit_lock_io` — wait for a bit to be cleared, when wanting to set it

## Synopsis

```
int wait_on_bit_lock_io (unsigned long * word, int bit, unsigned mode);
```

## Arguments

*word* the word being waited on, a kernel virtual address

*bit* the bit of the word being waited on

*mode* the task state to sleep in

## Description

Use the standard hashed waitqueue table to wait for a bit to be cleared and then to atomically set it. This is similar to `wait_on_bit`, but calls `io_schedule` instead of `schedule` for the actual waiting.

Returns zero if the bit was (eventually) found to be clear and was set. Returns non-zero if a signal was delivered to the process and the *mode* allows that signal to wake the process.

## Name

`wait_on_bit_lock_action` — wait for a bit to be cleared, when wanting to set it

## Synopsis

```
int wait_on_bit_lock_action (unsigned long * word, int bit,
wait_bit_action_f * action, unsigned mode);
```

## Arguments

*word*     the word being waited on, a kernel virtual address

*bit*     the bit of the word being waited on

*action*   the function used to sleep, which may take special actions

*mode*     the task state to sleep in

## Description

Use the standard hashed waitqueue table to wait for a bit to be cleared and then to set it, and allow the waiting action to be specified. This is like `wait_on_bit` but allows fine control of how the waiting is done.

Returns zero if the bit was (eventually) found to be clear and was set. Returns non-zero if a signal was delivered to the process and the *mode* allows that signal to wake the process.

## Name

`wait_on_atomic_t` — Wait for an `atomic_t` to become 0

## Synopsis

```
int wait_on_atomic_t (atomic_t * val, int (*action) (atomic_t *),
unsigned mode);
```

## Arguments

*val*        The atomic value being waited on, a kernel virtual address

*action*    the function used to sleep, which may take special actions

*mode*       the task state to sleep in

## Description

Wait for an `atomic_t` to become 0. We abuse the bit-wait waitqueue table for the purpose of getting a waitqueue, but we set the key to a bit number outside of the target 'word'.

## Name

`__wake_up` — wake up threads blocked on a waitqueue.

## Synopsis

```
void __wake_up (wait_queue_head_t * q, unsigned int mode, int
nr_exclusive, void * key);
```

## Arguments

<i>q</i>	the waitqueue
<i>mode</i>	which threads
<i>nr_exclusive</i>	how many wake-one or wake-many threads to wake up
<i>key</i>	is directly passed to the wakeup function

## Description

It may be assumed that this function implies a write memory barrier before changing the task state if and only if any tasks are woken up.

## Name

`__wake_up_sync_key` — wake up threads blocked on a waitqueue.

## Synopsis

```
void __wake_up_sync_key (wait_queue_head_t * q, unsigned int mode, int
nr_exclusive, void * key);
```

## Arguments

<i>q</i>	the waitqueue
<i>mode</i>	which threads
<i>nr_exclusive</i>	how many wake-one or wake-many threads to wake up
<i>key</i>	opaque value to be passed to wakeup targets

## Description

The sync wakeup differs that the waker knows that it will schedule away soon, so while the target thread will be woken up, it will not be migrated to another CPU - ie. the two threads are 'synchronized' with each other. This can prevent needless bouncing between CPUs.

On UP it can prevent extra preemption.

It may be assumed that this function implies a write memory barrier before changing the task state if and only if any tasks are woken up.

## Name

`finish_wait` — clean up after waiting in a queue

## Synopsis

```
void finish_wait (wait_queue_head_t * q, wait_queue_t * wait);
```

## Arguments

*q*       waitqueue waited on

*wait*    wait descriptor

## Description

Sets current thread back to running state and removes the wait descriptor from the given waitqueue if still queued.

## Name

`abort_exclusive_wait` — abort exclusive waiting in a queue

## Synopsis

```
void abort_exclusive_wait (wait_queue_head_t * q, wait_queue_t * wait,  
unsigned int mode, void * key);
```

## Arguments

*q*       waitqueue waited on

*wait*    wait descriptor

*mode*    runstate of the waiter to be woken

*key*     key to identify a wait bit queue or NULL

## Description

Sets current thread back to running state and removes the wait descriptor from the given waitqueue if still queued.

Wakes up the next waiter if the caller is concurrently woken up through the queue.

This prevents waiter starvation where an exclusive waiter aborts and is woken up concurrently and no one wakes up the next waiter.

## Name

`wake_up_bit` — wake up a waiter on a bit

## Synopsis

```
void wake_up_bit (void * word, int bit);
```

## Arguments

*word* the word being waited on, a kernel virtual address

*bit* the bit of the word being waited on

## Description

There is a standard hashed waitqueue table for generic use. This is the part of the hashtable's accessor API that wakes up waiters on a bit. For instance, if one were to have waiters on a bitflag, one would call `wake_up_bit` after clearing the bit.

In order for this to function properly, as it uses `waitqueue_active` internally, some kind of memory barrier must be done prior to calling this. Typically, this will be `smp_mb__after_atomic`, but in some cases where bitflags are manipulated non-atomically under a lock, one may need to use a less regular barrier, such as `fs/inode.c`'s `smp_mb`, because `spin_unlock` does not guarantee a memory barrier.



## Name

`wake_up_atomic_t` — Wake up a waiter on a `atomic_t`

## Synopsis

```
void wake_up_atomic_t (atomic_t * p);
```

## Arguments

*p* The `atomic_t` being waited on, a kernel virtual address

## Description

Wake up anyone waiting for the `atomic_t` to go to zero.

Abuse the bit-waker function and its waitqueue hash table set (the `atomic_t` check is done by the waiter's wake function, not the by the waker itself).

# High-resolution timers

## Name

`ktime_set` — Set a `ktime_t` variable from a seconds/nanoseconds value

## Synopsis

```
ktime_t ktime_set (const s64 secs, const unsigned long nsecs);
```

## Arguments

*secs*     seconds to set

*nsecs*    nanoseconds to set

## Return

The `ktime_t` representation of the value.

## Name

`ktime_equal` — Compares two `ktime_t` variables to see if they are equal

## Synopsis

```
int ktime_equal (const ktime_t cmp1, const ktime_t cmp2);
```

## Arguments

*cmp1*   comparable1

*cmp2*   comparable2

## Description

Compare two `ktime_t` variables.

## Return

1 if equal.

## Name

`ktime_compare` — Compares two `ktime_t` variables for less, greater or equal

## Synopsis

```
int ktime_compare (const ktime_t cmp1, const ktime_t cmp2);
```

## Arguments

*cmp1*   comparable1

*cmp2*   comparable2

## Return

... `cmp1 < cmp2`: return `<0` `cmp1 == cmp2`: return `0` `cmp1 > cmp2`: return `>0`

## Name

`ktime_after` — Compare if a `ktime_t` value is bigger than another one.

## Synopsis

```
bool ktime_after (const ktime_t cmp1, const ktime_t cmp2);
```

## Arguments

*cmp1*   comparable1

*cmp2*   comparable2

## Return

true if *cmp1* happened after *cmp2*.

## Name

`ktime_before` — Compare if a `ktime_t` value is smaller than another one.

## Synopsis

```
bool ktime_before (const ktime_t cmp1, const ktime_t cmp2);
```

## Arguments

*cmp1*   comparable1

*cmp2*   comparable2

## Return

true if `cmp1` happened before `cmp2`.

## Name

`ktime_to_timespec_cond` — convert a `ktime_t` variable to `timespec` format only if the variable contains data

## Synopsis

```
bool ktime_to_timespec_cond (const ktime_t kt, struct timespec * ts);
```

## Arguments

*kt* the `ktime_t` variable to convert

*ts* the `timespec` variable to store the result in

## Return

true if there was a successful conversion, false if `kt` was 0.

## Name

`ktime_to_timespec64_cond` — convert a `ktime_t` variable to `timespec64` format only if the variable contains data

## Synopsis

```
bool ktime_to_timespec64_cond (const ktime_t kt, struct timespec64 *
ts);
```

## Arguments

*kt* the `ktime_t` variable to convert

*ts* the `timespec` variable to store the result in

## Return

true if there was a successful conversion, false if `kt` was 0.



## Name

struct hrtimer — the basic hrtimer structure

## Synopsis

```
struct hrtimer {
    struct timerqueue_node node;
    ktime_t _softexpires;
    enum hrtimer_restart (* function) (struct hrtimer *);
    struct hrtimer_clock_base * base;
    u8 state;
    u8 is_rel;
#ifdef CONFIG_TIMER_STATS
    int start_pid;
    void * start_site;
    char start_comm[16];
#endif
};
```

## Members

node	timerqueue node, which also manages node.expires, the absolute expiry time in the hrtimers internal representation. The time is related to the clock on which the timer is based. Is setup by adding slack to the _softexpires value. For non range timers identical to _softexpires.
_softexpires	the absolute earliest expiry time of the hrtimer. The time which was given as expiry time when the timer was armed.
function	timer expiry callback function
base	pointer to the timer base (per cpu and per clock)
state	state information (See bit values above)
is_rel	Set if the timer was armed relative
start_pid	timer statistics field to store the pid of the task which started the timer
start_site	timer statistics field to store the site where the timer was started
start_comm[16]	timer statistics field to store the name of the process which started the timer

## Description

The hrtimer structure must be initialized by `hrtimer_init`

## Name

struct hrtimer\_sleeper — simple sleeper structure

## Synopsis

```
struct hrtimer_sleeper {  
    struct hrtimer timer;  
    struct task_struct * task;  
};
```

## Members

timer      embedded timer structure

task      task to wake up

## Description

task is set to NULL, when the timer expires.

## Name

struct hrtimer\_clock\_base — the timer base for a specific clock

## Synopsis

```
struct hrtimer_clock_base {  
    struct hrtimer_cpu_base * cpu_base;  
    int index;  
    clockid_t clockid;  
    struct timerqueue_head active;  
    ktime_t (* get_time) (void);  
    ktime_t offset;  
};
```

## Members

cpu_base	per cpu clock base
index	clock type index for per_cpu support when moving a timer to a base on another cpu.
clockid	clock id for per_cpu support
active	red black tree root node for the active timers
get_time	function to retrieve the current time of the clock
offset	offset of this clock to the monotonic base

## Name

`hrtimer_start` — (re)start an hrtimer on the current CPU

## Synopsis

```
void hrtimer_start (struct hrtimer * timer, ktime_t tim, const enum  
hrtimer_mode mode);
```

## Arguments

*timer*    the timer to be added

*tim*      expiry time

*mode*    expiry mode: absolute (`HRTIMER_MODE_ABS`) or relative (`HRTIMER_MODE_REL`)

## Name

`hrtimer_forward_now` — forward the timer expiry so it expires after now

## Synopsis

```
u64 hrtimer_forward_now (struct hrtimer * timer, ktime_t interval);
```

## Arguments

*timer*        hrtimer to forward

*interval*    the interval to forward

## Description

Forward the timer expiry so it will expire after the current time of the hrtimer clock base. Returns the number of overruns.

Can be safely called from the callback function of *timer*. If called from other contexts *timer* must neither be enqueued nor running the callback and the caller needs to take care of serialization.

## Note

This only updates the timer expiry value and does not requeue the timer.

## Name

`hrtimer_forward` — forward the timer expiry

## Synopsis

```
u64 hrtimer_forward (struct hrtimer * timer, ktime_t now, ktime_t
interval);
```

## Arguments

*timer*        hrtimer to forward

*now*          forward past this time

*interval*    the interval to forward

## Description

Forward the timer expiry so it will expire in the future. Returns the number of overruns.

Can be safely called from the callback function of *timer*. If called from other contexts *timer* must neither be enqueued nor running the callback and the caller needs to take care of serialization.

## Note

This only updates the timer expiry value and does not requeue the timer.

## Name

`hrtimer_start_range_ns` — (re)start an hrtimer on the current CPU

## Synopsis

```
void hrtimer_start_range_ns (struct hrtimer * timer, ktime_t tim,  
unsigned long delta_ns, const enum hrtimer_mode mode);
```

## Arguments

*timer*        the timer to be added

*tim*         expiry time

*delta\_ns*    "slack" range for the timer

*mode*        expiry mode: absolute (HRTIMER\_MODE\_ABS) or relative (HRTIMER\_MODE\_REL)

## Name

`hrtimer_try_to_cancel` — try to deactivate a timer

## Synopsis

```
int hrtimer_try_to_cancel (struct hrtimer * timer);
```

## Arguments

*timer*    hrtimer to stop

## Returns

0 when the timer was not active 1 when the timer was active -1 when the timer is currently excuting the callback function and cannot be stopped



## Name

`hrtimer_cancel` — cancel a timer and wait for the handler to finish.

## Synopsis

```
int hrtimer_cancel (struct hrtimer * timer);
```

## Arguments

*timer* the timer to be cancelled

## Returns

0 when the timer was not active 1 when the timer was active

## Name

`__hrtimer_get_remaining` — get remaining time for the timer

## Synopsis

```
ktime_t __hrtimer_get_remaining (const struct hrtimer * timer, bool  
adjust);
```

## Arguments

*timer*    the timer to read

*adjust*   adjust relative timers when CONFIG\_TIME\_LOW\_RES=y

## Name

`hrtimer_init` — initialize a timer to the given clock

## Synopsis

```
void hrtimer_init (struct hrtimer * timer, clockid_t clock_id, enum  
hrtimer_mode mode);
```

## Arguments

*timer*        the timer to be initialized

*clock\_id*    the clock to be used

*mode*        timer mode abs/rel

## Name

`schedule_hrttimeout_range` — sleep until timeout

## Synopsis

```
int __sched schedule_hrttimeout_range (ktime_t * expires, unsigned long
delta, const enum hrtimer_mode mode);
```

## Arguments

*expires*    timeout value (ktime\_t)

*delta*        slack in expires timeout (ktime\_t)

*mode*        timer mode, HRTIMER\_MODE\_ABS or HRTIMER\_MODE\_REL

## Description

Make the current task sleep until the given expiry time has elapsed. The routine will return immediately unless the current task state has been set (see `set_current_state`).

The *delta* argument gives the kernel the freedom to schedule the actual wakeup to a time that is both power and performance friendly. The kernel give the normal best effort behavior for “*expires+delta*”, but may decide to fire the timer earlier, but no earlier than *expires*.

You can set the task state as follows -

`TASK_UNINTERRUPTIBLE` - at least *timeout* time is guaranteed to pass before the routine returns.

`TASK_INTERRUPTIBLE` - the routine may return early if a signal is delivered to the current task.

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

Returns 0 when the timer has expired otherwise -EINTR

## Name

`schedule_hrtimeout` — sleep until timeout

## Synopsis

```
int __sched schedule_hrtimeout (ktime_t * expires, const enum
hrtimer_mode mode);
```

## Arguments

*expires* timeout value (ktime\_t)

*mode* timer mode, HRTIMER\_MODE\_ABS or HRTIMER\_MODE\_REL

## Description

Make the current task sleep until the given expiry time has elapsed. The routine will return immediately unless the current task state has been set (see `set_current_state`).

You can set the task state as follows -

`TASK_UNINTERRUPTIBLE` - at least *timeout* time is guaranteed to pass before the routine returns.

`TASK_INTERRUPTIBLE` - the routine may return early if a signal is delivered to the current task.

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

Returns 0 when the timer has expired otherwise `-EINTR`

## Workqueues and Kevents

## Name

`work_pending` — Find out whether a work item is currently pending

## Synopsis

```
work_pending ( work );
```

## Arguments

*work*    The work item in question

## Name

`delayed_work_pending` — Find out whether a delayable work item is currently pending

## Synopsis

```
delayed_work_pending ( w );
```

## Arguments

*w* The work item in question

## Name

`alloc_workqueue` — allocate a workqueue

## Synopsis

```
alloc_workqueue ( fmt, flags, max_active, args... );
```

## Arguments

<i>fmt</i>	printf format for the name of the workqueue
<i>flags</i>	WQ_* flags
<i>max_active</i>	max in-flight work items, 0 for default <i>args...</i> : args for <i>fmt</i>
<i>args...</i>	variable arguments

## Description

Allocate a workqueue with the specified parameters. For detailed information on WQ\_\* flags, please refer to Documentation/workqueue.txt.

The `__lock_name` macro dance is to guarantee that single `lock_class_key` doesn't end up with different namesm, which isn't allowed by lockdep.

## RETURNS

Pointer to the allocated workqueue on success, NULL on failure.



## Name

`alloc_ordered_workqueue` — allocate an ordered workqueue

## Synopsis

```
alloc_ordered_workqueue ( fmt, flags, args... );
```

## Arguments

*fmt*            printf format for the name of the workqueue

*flags*        WQ\_\* flags (only WQ\_FREEZABLE and WQ\_MEM\_RECLAIM are meaningful) *args...*:  
args for *fmt*

*args...*    variable arguments

## Description

Allocate an ordered workqueue. An ordered workqueue executes at most one work item at any given time in the queued order. They are implemented as unbound workqueues with *max\_active* of one.

## RETURNS

Pointer to the allocated workqueue on success, NULL on failure.

## Name

`queue_work` — queue work on a workqueue

## Synopsis

```
bool queue_work (struct workqueue_struct * wq, struct work_struct *  
work);
```

## Arguments

*wq*      workqueue to use

*work*    work to queue

## Description

Returns `false` if *work* was already on a queue, `true` otherwise.

We queue the work to the CPU on which it was submitted, but if the CPU dies it can be processed by another CPU.

## Name

`queue_delayed_work` — queue work on a workqueue after delay

## Synopsis

```
bool queue_delayed_work (struct workqueue_struct * wq, struct
delayed_work * dwork, unsigned long delay);
```

## Arguments

*wq*        workqueue to use

*dwork*    delayable work to queue

*delay*    number of jiffies to wait before queueing

## Description

Equivalent to `queue_delayed_work_on` but tries to use the local CPU.

## Name

`mod_delayed_work` — modify delay of or queue a delayed work

## Synopsis

```
bool mod_delayed_work (struct workqueue_struct * wq, struct delayed_work  
* dwork, unsigned long delay);
```

## Arguments

*wq*        workqueue to use

*dwork*    work to queue

*delay*    number of jiffies to wait before queueing

## Description

`mod_delayed_work_on` on local CPU.

## Name

`schedule_work_on` — put work task on a specific cpu

## Synopsis

```
bool schedule_work_on (int cpu, struct work_struct * work);
```

## Arguments

*cpu*     cpu to put the work task on

*work*    job to be done

## Description

This puts a job on a specific cpu

## Name

`schedule_work` — put work task in global workqueue

## Synopsis

```
bool schedule_work (struct work_struct * work);
```

## Arguments

*work* job to be done

## Description

Returns `false` if *work* was already on the kernel-global workqueue and `true` otherwise.

This puts a job in the kernel-global workqueue if it was not already queued and leaves it in the same position on the kernel-global workqueue otherwise.

## Name

`flush_scheduled_work` — ensure that any scheduled work has run to completion.

## Synopsis

```
void flush_scheduled_work ( void );
```

## Arguments

*void* no arguments

## Description

Forces execution of the kernel-global workqueue and blocks until its completion.

Think twice before calling this function! It's very easy to get into trouble if you don't take great care. Either of the following situations

### will lead to deadlock

One of the work items currently on the workqueue needs to acquire a lock held by your code or its caller.

Your code is running in the context of a work routine.

They will be detected by lockdep when they occur, but the first might not occur very often. It depends on what work items are on the workqueue and what locks they need, which you have no control over.

In most situations flushing the entire workqueue is overkill; you merely need to know that a particular work item isn't queued and isn't running. In such cases you should use `cancel_delayed_work_sync` or `cancel_work_sync` instead.

## Name

`schedule_delayed_work_on` — queue work in global workqueue on CPU after delay

## Synopsis

```
bool schedule_delayed_work_on (int cpu, struct delayed_work * dwork,  
unsigned long delay);
```

## Arguments

*cpu*      cpu to use

*dwork*    job to be done

*delay*    number of jiffies to wait

## Description

After waiting for a given time this puts a job in the kernel-global workqueue on the specified CPU.



## Name

`schedule_delayed_work` — put work task in global workqueue after delay

## Synopsis

```
bool schedule_delayed_work (struct delayed_work * dwork, unsigned long  
delay);
```

## Arguments

*dwork*    job to be done

*delay*    number of jiffies to wait or 0 for immediate execution

## Description

After waiting for a given time this puts a job in the kernel-global workqueue.

## Name

keventd\_up — is workqueue initialized yet?

## Synopsis

```
bool keventd_up ( void );
```

## Arguments

*void* no arguments

## Name

`queue_work_on` — queue work on specific cpu

## Synopsis

```
bool queue_work_on (int cpu, struct workqueue_struct * wq, struct
work_struct * work);
```

## Arguments

*cpu*     CPU number to execute work on

*wq*     workqueue to use

*work*   work to queue

## Description

We queue the work to a specific CPU, the caller must ensure it can't go away.

## Return

false if *work* was already on a queue, true otherwise.

## Name

`queue_delayed_work_on` — queue work on specific CPU after delay

## Synopsis

```
bool queue_delayed_work_on (int cpu, struct workqueue_struct * wq,
struct delayed_work * dwork, unsigned long delay);
```

## Arguments

*cpu*      CPU number to execute work on

*wq*        workqueue to use

*dwork*    work to queue

*delay*    number of jiffies to wait before queueing

## Return

false if *work* was already on a queue, true otherwise. If *delay* is zero and *dwork* is idle, it will be scheduled for immediate execution.

## Name

`mod_delayed_work_on` — modify delay of or queue a delayed work on specific CPU

## Synopsis

```
bool mod_delayed_work_on (int cpu, struct workqueue_struct * wq, struct
delayed_work * dwork, unsigned long delay);
```

## Arguments

*cpu*      CPU number to execute work on

*wq*        workqueue to use

*dwork*    work to queue

*delay*    number of jiffies to wait before queueing

## Description

If *dwork* is idle, equivalent to `queue_delayed_work_on`; otherwise, modify *dwork*'s timer so that it expires after *delay*. If *delay* is zero, *work* is guaranteed to be scheduled immediately regardless of its current state.

## Return

false if *dwork* was idle and queued, true if *dwork* was pending and its timer was modified.

This function is safe to call from any context including IRQ handler. See `try_to_grab_pending` for details.

## Name

`flush_workqueue` — ensure that any scheduled work has run to completion.

## Synopsis

```
void flush_workqueue (struct workqueue_struct * wq);
```

## Arguments

*wq*   workqueue to flush

## Description

This function sleeps until all work items which were queued on entry have finished execution, but it is not livelocked by new incoming ones.

## Name

`drain_workqueue` — drain a workqueue

## Synopsis

```
void drain_workqueue (struct workqueue_struct * wq);
```

## Arguments

*wq*   workqueue to drain

## Description

Wait until the workqueue becomes empty. While draining is in progress, only chain queueing is allowed. IOW, only currently pending or running work items on *wq* can queue further work items on it. *wq* is flushed repeatedly until it becomes empty. The number of flushing is determined by the depth of chaining and should be relatively short. Whine if it takes too long.

## Name

`flush_work` — wait for a work to finish executing the last queueing instance

## Synopsis

```
bool flush_work (struct work_struct * work);
```

## Arguments

*work* the work to flush

## Description

Wait until *work* has finished execution. *work* is guaranteed to be idle on return if it hasn't been requeued since flush started.

## Return

true if `flush_work` waited for the work to finish execution, false if it was already idle.



## Name

`cancel_work_sync` — cancel a work and wait for it to finish

## Synopsis

```
bool cancel_work_sync (struct work_struct * work);
```

## Arguments

*work* the work to cancel

## Description

Cancel *work* and wait for its execution to finish. This function can be used even if the work re-queues itself or migrates to another workqueue. On return from this function, *work* is guaranteed to be not pending or executing on any CPU.

`cancel_work_sync(delayed_work->work)` must not be used for `delayed_work`'s. Use `cancel_delayed_work_sync` instead.

The caller must ensure that the workqueue on which *work* was last queued can't be destroyed before this function returns.

## Return

true if *work* was pending, false otherwise.

## Name

`flush_delayed_work` — wait for a `dwork` to finish executing the last queueing

## Synopsis

```
bool flush_delayed_work (struct delayed_work * dwork);
```

## Arguments

*dwork* the delayed work to flush

## Description

Delayed timer is cancelled and the pending work is queued for immediate execution. Like `flush_work`, this function only considers the last queueing instance of *dwork*.

## Return

true if `flush_work` waited for the work to finish execution, false if it was already idle.

## Name

`cancel_delayed_work` — cancel a delayed work

## Synopsis

```
bool cancel_delayed_work (struct delayed_work * dwork);
```

## Arguments

*dwork*   delayed\_work to cancel

## Description

Kill off a pending `delayed_work`.

## Return

`true` if *dwork* was pending and canceled; `false` if it wasn't pending.

## Note

The work callback function may still be running on return, unless it returns `true` and the work doesn't re-arm itself. Explicitly flush or use `cancel_delayed_work_sync` to wait on it.

This function is safe to call from any context including IRQ handler.

## Name

`cancel_delayed_work_sync` — cancel a delayed work and wait for it to finish

## Synopsis

```
bool cancel_delayed_work_sync (struct delayed_work * dwork);
```

## Arguments

*dwork* the delayed work cancel

## Description

This is `cancel_work_sync` for delayed works.

## Return

true if *dwork* was pending, false otherwise.

## Name

`execute_in_process_context` — reliably execute the routine with user context

## Synopsis

```
int execute_in_process_context (work_func_t fn, struct execute_work *  
ew);
```

## Arguments

*fn* the function to execute

*ew* guaranteed storage for the execute work structure (must be available when the work executes)

## Description

Executes the function immediately if process context is available, otherwise schedules the function for delayed execution.

## Return

0 - function was executed 1 - function was scheduled for execution

## Name

`destroy_workqueue` — safely terminate a workqueue

## Synopsis

```
void destroy_workqueue (struct workqueue_struct * wq);
```

## Arguments

*wq* target workqueue

## Description

Safely destroy a workqueue. All work currently pending will be done first.

## Name

`workqueue_set_max_active` — adjust `max_active` of a workqueue

## Synopsis

```
void workqueue_set_max_active (struct workqueue_struct * wq, int  
max_active);
```

## Arguments

*wq*                    target workqueue

*max\_active*    new `max_active` value.

## Description

Set `max_active` of *wq* to *max\_active*.

## CONTEXT

Don't call from IRQ context.

## Name

`workqueue_congested` — test whether a workqueue is congested

## Synopsis

```
bool workqueue_congested (int cpu, struct workqueue_struct * wq);
```

## Arguments

*cpu*    CPU in question

*wq*     target workqueue

## Description

Test whether *wq*'s cpu workqueue for *cpu* is congested. There is no synchronization around this function and the test result is unreliable and only useful as advisory hints or for debugging.

If *cpu* is `WORK_CPU_UNBOUND`, the test is performed on the local CPU. Note that both per-cpu and unbound workqueues may be associated with multiple `pool_workqueues` which have separate congested states. A workqueue being congested on one CPU doesn't mean the workqueue is also congested on other CPUs / NUMA nodes.

## Return

true if congested, false otherwise.



## Name

`work_busy` — test whether a work is currently pending or running

## Synopsis

```
unsigned int work_busy (struct work_struct * work);
```

## Arguments

*work* the work to be tested

## Description

Test whether *work* is currently pending or running. There is no synchronization around this function and the test result is unreliable and only useful as advisory hints or for debugging.

## Return

OR'd bitmask of `WORK_BUSY_*` bits.

## Name

`work_on_cpu` — run a function in user context on a particular cpu

## Synopsis

```
long work_on_cpu (int cpu, long (*fn) (void *), void * arg);
```

## Arguments

*cpu* the cpu to run on

*fn* the function to run

*arg* the function arg

## Description

It is up to the caller to ensure that the cpu doesn't go offline. The caller must not hold any locks which would prevent *fn* from completing.

## Return

The value *fn* returns.

## Internal Functions

## Name

`wait_task_stopped` — Wait for `TASK_STOPPED` or `TASK_TRACED`

## Synopsis

```
int wait_task_stopped (struct wait_opts * wo, int ptrace, struct
task_struct * p);
```

## Arguments

*wo*        wait options

*ptrace*    is the wait for ptrace

*p*        task to wait for

## Description

Handle `sys_wait4` work for `p` in state `TASK_STOPPED` or `TASK_TRACED`.

## CONTEXT

`read_lock(tasklist_lock)`, which is released if return value is non-zero. Also, grabs and releases `p->sighand->siglock`.

## RETURNS

0 if wait condition didn't exist and search for other wait conditions should continue. Non-zero return, -`errno` on failure and `p`'s pid on success, implies that `tasklist_lock` is released and wait condition search should terminate.

## Name

`task_set_jobctl_pending` — set jobctl pending bits

## Synopsis

```
bool task_set_jobctl_pending (struct task_struct * task, unsigned long  
mask);
```

## Arguments

*task* target task

*mask* pending bits to set

## Description

Clear *mask* from *task->jobctl*. *mask* must be subset of `JOBCTL_PENDING_MASK | JOBCTL_STOP_CONSUME | JOBCTL_STOP_SIGMASK | JOBCTL_TRAPPING`. If stop signo is being set, the existing signo is cleared. If *task* is already being killed or exiting, this function becomes noop.

## CONTEXT

Must be called with *task->sighand->siglock* held.

## RETURNS

true if *mask* is set, false if made noop because *task* was dying.

## Name

`task_clear_jobctl_trapping` — clear jobctl trapping bit

## Synopsis

```
void task_clear_jobctl_trapping (struct task_struct * task);
```

## Arguments

*task* target task

## Description

If `JOBCTL_TRAPPING` is set, a ptracer is waiting for us to enter `TRACED`. Clear it and wake up the ptracer. Note that we don't need any further locking. *task*->siglock guarantees that *task*->parent points to the ptracer.

## CONTEXT

Must be called with *task*->sigband->siglock held.

## Name

`task_clear_jobctl_pending` — clear jobctl pending bits

## Synopsis

```
void task_clear_jobctl_pending (struct task_struct * task, unsigned long  
mask);
```

## Arguments

*task* target task

*mask* pending bits to clear

## Description

Clear *mask* from *task->jobctl*. *mask* must be subset of `JOBCTL_PENDING_MASK`. If `JOBCTL_STOP_PENDING` is being cleared, other `STOP` bits are cleared together.

If clearing of *mask* leaves no stop or trap pending, this function calls `task_clear_jobctl_trapping`.

## CONTEXT

Must be called with *task->sighand->siglock* held.

## Name

`task_participate_group_stop` — participate in a group stop

## Synopsis

```
bool task_participate_group_stop (struct task_struct * task);
```

## Arguments

*task* task participating in a group stop

## Description

*task* has `JOBCTL_STOP_PENDING` set and is participating in a group stop. Group stop states are cleared and the group stop count is consumed if `JOBCTL_STOP_CONSUME` was set. If the consumption completes the group stop, the appropriate `SIGNAL_*` flags are set.

## CONTEXT

Must be called with *task*->sigband->siglock held.

## RETURNS

`true` if group stop completion should be notified to the parent, `false` otherwise.

## Name

`ptrace_trap_notify` — schedule trap to notify ptracer

## Synopsis

```
void ptrace_trap_notify (struct task_struct * t);
```

## Arguments

`t` tracee wanting to notify tracer

## Description

This function schedules sticky ptrace trap which is cleared on the next `TRAP_STOP` to notify ptracer of an event. `t` must have been seized by ptracer.

If `t` is running, `STOP` trap will be taken. If trapped for `STOP` and ptracer is listening for events, tracee is woken up so that it can re-trap for the new event. If trapped otherwise, `STOP` trap will be eventually taken without returning to userland after the existing traps are finished by `PTRACE_CONT`.

## CONTEXT

Must be called with `task->sigband->siglock` held.



## Name

`do_notify_parent_cldstop` — notify parent of stopped/continued state change

## Synopsis

```
void do_notify_parent_cldstop (struct task_struct * tsk, bool
for_ptracer, int why);
```

## Arguments

<i>tsk</i>	task reporting the state change
<i>for_ptracer</i>	the notification is for ptracer
<i>why</i>	CLD_{CONTINUED STOPPED TRAPPED} to report

## Description

Notify *tsk*'s parent that the stopped/continued state has changed. If *for\_ptracer* is false, *tsk*'s group leader notifies to its real parent. If true, *tsk* reports to *tsk*->parent which should be the ptracer.

## CONTEXT

Must be called with `tasklist_lock` at least read locked.

## Name

`do_signal_stop` — handle group stop for SIGSTOP and other stop signals

## Synopsis

```
bool do_signal_stop (int signr);
```

## Arguments

*signr*    *signr* causing group stop if initiating

## Description

If `JOBCTL_STOP_PENDING` is not set yet, initiate group stop with *signr* and participate in it. If already set, participate in the existing group stop. If participated in a group stop (and thus slept), `true` is returned with siglock released.

If ptraced, this function doesn't handle stop itself. Instead, `JOBCTL_TRAP_STOP` is scheduled and `false` is returned with siglock untouched. The caller must ensure that INTERRUPT trap handling takes places afterwards.

## CONTEXT

Must be called with *current->sigband->siglock* held, which is released on `true` return.

## RETURNS

`false` if group stop is already cancelled or ptrace trap is scheduled. `true` if participated in group stop.

## Name

`do_jobctl_trap` — take care of ptrace jobctl traps

## Synopsis

```
void do_jobctl_trap ( void );
```

## Arguments

*void* no arguments

## Description

When `PT_SEIZED`, it's used for both group stop and explicit `SEIZE/INTERRUPT` traps. Both generate `PTRACE_EVENT_STOP` trap with accompanying `siginfo`. If stopped, lower eight bits of `exit_code` contain the stop signal; otherwise, `SIGTRAP`.

When `!PT_SEIZED`, it's used only for group stop trap with stop signal number as `exit_code` and no `siginfo`.

## CONTEXT

Must be called with `current->sigband->siglock` held, which may be released and re-acquired before returning with intervening sleep.

## Name

`signal_delivered` —

## Synopsis

```
void signal_delivered (struct ksignal * ksig, int stepping);
```

## Arguments

*ksig*            kernel signal struct

*stepping*      nonzero if debugger single-step or block-step in use

## Description

This function should be called when a signal has successfully been delivered. It updates the blocked signals accordingly (*ksig*->ka.sa.sa\_mask is always blocked, and the signal itself is blocked unless SA\_NODEFER is set in *ksig*->ka.sa.sa\_flags. Tracing is notified.

## Name

`sys_restart_syscall` — restart a system call

## Synopsis

```
long sys_restart_syscall ( void );
```

## Arguments

*void* no arguments

## Name

`set_current_blocked` — change `current->blocked` mask

## Synopsis

```
void set_current_blocked (sigset_t * newset);
```

## Arguments

*newset*    new mask

## Description

It is wrong to change `->blocked` directly, this helper should be used to ensure the process can't miss a shared signal we are going to block.

## Name

`sys_rt_sigprocmask` — change the list of currently blocked signals

## Synopsis

```
long sys_rt_sigprocmask (int how, sigset_t __user * nset, sigset_t  
__user * oset, size_t sigsetsize);
```

## Arguments

<i>how</i>	whether to add, remove, or set signals
<i>nset</i>	stores pending signals
<i>oset</i>	previous value of signal mask if non-null
<i>sigsetsize</i>	size of sigset_t type

## Name

`sys_rt_sigpending` — examine a pending signal that has been raised while blocked

## Synopsis

```
long sys_rt_sigpending (sigset_t __user * uset, size_t sigsetsize);
```

## Arguments

*uset*                stores pending signals

*sigsetsize*        size of `sigset_t` type or larger



## Name

`do_sigtimedwait` — wait for queued signals specified in *which*

## Synopsis

```
int do_sigtimedwait (const sigset_t * which, siginfo_t * info, const
struct timespec * ts);
```

## Arguments

*which* queued signals to wait for

*info* if non-null, the signal's siginfo is returned here

*ts* upper bound on process time suspension

## Name

`sys_rt_sigtimedwait` — synchronously wait for queued signals specified in *uthese*

## Synopsis

```
long sys_rt_sigtimedwait (const sigset_t __user * uthese, siginfo_t  
__user * uinfo, const struct timespec __user * uts, size_t sigsetsize);
```

## Arguments

<i>uthese</i>	queued signals to wait for
<i>uinfo</i>	if non-null, the signal's siginfo is returned here
<i>uts</i>	upper bound on process time suspension
<i>sigsetsize</i>	size of sigset_t type

## Name

`sys_kill` — send a signal to a process

## Synopsis

```
long sys_kill (pid_t pid, int sig);
```

## Arguments

*pid* the PID of the process

*sig* signal to be sent

## Name

`sys_tgkill` — send signal to one specific thread

## Synopsis

```
long sys_tgkill (pid_t tgid, pid_t pid, int sig);
```

## Arguments

*tgid* the thread group ID of the thread

*pid* the PID of the thread

*sig* signal to be sent

## Description

This syscall also checks the *tgid* and returns `-ESRCH` even if the PID exists but it's not belonging to the target process anymore. This method solves the problem of threads exiting and PIDs getting reused.

## Name

`sys_kill` — send signal to one specific task

## Synopsis

```
long sys_kill (pid_t pid, int sig);
```

## Arguments

*pid* the PID of the task

*sig* signal to be sent

## Description

Send a signal to only one task, even if it's a `CLONE_THREAD` task.

## Name

`sys_rt_sigqueueinfo` — send signal information to a signal

## Synopsis

```
long sys_rt_sigqueueinfo (pid_t pid, int sig, siginfo_t __user * uinfo);
```

## Arguments

*pid*      the PID of the thread

*sig*      signal to be sent

*uinfo*    signal info to be sent

## Name

`sys_sigpending` — examine pending signals

## Synopsis

```
long sys_sigpending (old_sigset_t __user * set);
```

## Arguments

*set* where mask of pending signal is returned

## Name

`sys_sigprocmask` — examine and change blocked signals

## Synopsis

```
long sys_sigprocmask (int how, old_sigset_t __user * nset, old_sigset_t  
__user * oset);
```

## Arguments

*how*    whether to add, remove, or set signals

*nset*    signals to add or remove (if non-null)

*oset*    previous value of signal mask if non-null

## Description

Some platforms have their own version with special arguments; others support only `sys_rt_sigprocmask`.



## Name

`sys_rt_sigaction` — alter an action taken by a process

## Synopsis

```
long sys_rt_sigaction (int sig, const struct sigaction __user * act,  
struct sigaction __user * oact, size_t sigsetsize);
```

## Arguments

<i>sig</i>	signal to be sent
<i>act</i>	new sigaction
<i>oact</i>	used to save the previous sigaction
<i>sigsetsize</i>	size of sigset_t type

## Name

`sys_rt_sigsuspend` — replace the signal mask for a value with the *unewset* value until a signal is received

## Synopsis

```
long sys_rt_sigsuspend (sigset_t __user * unewset, size_t sigsetsize);
```

## Arguments

*unewset*        new signal mask value

*sigsetsize*    size of sigset\_t type

## Name

`kthread_run` — create and wake a thread.

## Synopsis

```
kthread_run ( threadfn, data, namefmt, ... );
```

## Arguments

*threadfn*    the function to run until `signal_pending(current)`.

*data*        data ptr for *threadfn*.

*namefmt*     printf-style name for the thread.

*...*        variable arguments

## Description

Convenient wrapper for `kthread_create` followed by `wake_up_process`. Returns the `kthread` or `ERR_PTR(-ENOMEM)`.

## Name

`kthread_should_stop` — should this kthread return now?

## Synopsis

```
bool kthread_should_stop ( void );
```

## Arguments

*void* no arguments

## Description

When someone calls `kthread_stop` on your kthread, it will be woken and this will return true. You should then return, and your return value will be passed through to `kthread_stop`.

## Name

`kthread_should_park` — should this kthread park now?

## Synopsis

```
bool kthread_should_park ( void );
```

## Arguments

*void* no arguments

## Description

When someone calls `kthread_park` on your kthread, it will be woken and this will return true. You should then do the necessary cleanup and call `kthread_parkme`.

Similar to `kthread_should_stop`, but this keeps the thread alive and in a park position. `kthread_unpark` “restarts” the thread and calls the thread function again.

## Name

`kthread_freezable_should_stop` — should this freezable kthread return now?

## Synopsis

```
bool kthread_freezable_should_stop (bool * was_frozen);
```

## Arguments

*was\_frozen* optional out parameter, indicates whether current was frozen

## Description

`kthread_should_stop` for freezable kthreads, which will enter refrigerator if necessary. This function is safe from `kthread_stop` / freezer deadlock and freezable kthreads should use this function instead of calling `try_to_freeze` directly.

## Name

`kthread_create_on_node` — create a kthread.

## Synopsis

```
struct task_struct * kthread_create_on_node (int (*threadfn) (void  
*data), void * data, int node, const char namefmt[], ...);
```

## Arguments

<i>threadfn</i>	the function to run until <code>signal_pending(current)</code> .
<i>data</i>	data ptr for <i>threadfn</i> .
<i>node</i>	task and thread structures for the thread are allocated on this node
<i>namefmt</i> [ ]	printf-style name for the thread.
...	variable arguments

## Description

This helper function creates and names a kernel thread. The thread will be stopped: use `wake_up_process` to start it. See also `kthread_run`. The new thread has `SCHED_NORMAL` policy and is affine to all CPUs.

If thread is going to be bound on a particular cpu, give its node in *node*, to get NUMA affinity for kthread stack, or else give `NUMA_NO_NODE`. When woken, the thread will run *threadfn*() with *data* as its argument. *threadfn*() can either call `do_exit` directly if it is a standalone thread for which no one will call `kthread_stop`, or return when '`kthread_should_stop`' is true (which means `kthread_stop` has been called). The return value should be zero or a negative error number; it will be passed to `kthread_stop`.

Returns a `task_struct` or `ERR_PTR(-ENOMEM)` or `ERR_PTR(-EINTR)`.

## Name

`kthread_bind` — bind a just-created kthread to a cpu.

## Synopsis

```
void kthread_bind (struct task_struct * p, unsigned int cpu);
```

## Arguments

*p*      thread created by `kthread_create`.

*cpu*    cpu (might not be online, must be possible) for *k* to run on.

## Description

This function is equivalent to `set_cpus_allowed`, except that *cpu* doesn't need to be online, and the thread must be stopped (i.e., just returned from `kthread_create`).



## Name

`kthread_unpark` — unpark a thread created by `kthread_create`.

## Synopsis

```
void kthread_unpark (struct task_struct * k);
```

## Arguments

*k* thread created by `kthread_create`.

## Description

Sets `kthread_should_park` for *k* to return false, wakes it, and waits for it to return. If the thread is marked `percpu` then its bound to the `cpu` again.

## Name

`kthread_park` — park a thread created by `kthread_create`.

## Synopsis

```
int kthread_park (struct task_struct * k);
```

## Arguments

*k* thread created by `kthread_create`.

## Description

Sets `kthread_should_park` for *k* to return true, wakes it, and waits for it to return. This can also be called after `kthread_create` instead of calling `wake_up_process`: the thread will park without calling `threadfn`.

Returns 0 if the thread is parked, `-ENOSYS` if the thread exited. If called by the kthread itself just the park bit is set.

## Name

`kthread_stop` — stop a thread created by `kthread_create`.

## Synopsis

```
int kthread_stop (struct task_struct * k);
```

## Arguments

*k* thread created by `kthread_create`.

## Description

Sets `kthread_should_stop` for *k* to return true, wakes it, and waits for it to exit. This can also be called after `kthread_create` instead of calling `wake_up_process`: the thread will exit without calling `threadfn`.

If `threadfn` may call `do_exit` itself, the caller must ensure `task_struct` can't go away.

Returns the result of `threadfn`, or `-EINTR` if `wake_up_process` was never called.

## Name

`kthread_worker_fn` — kthread function to process `kthread_worker`

## Synopsis

```
int kthread_worker_fn (void * worker_ptr);
```

## Arguments

*worker\_ptr* pointer to initialized `kthread_worker`

## Description

This function can be used as *threadfn* to `kthread_create` or `kthread_run` with *worker\_ptr* argument pointing to an initialized `kthread_worker`. The started kthread will process `work_list` until the it is stopped with `kthread_stop`. A kthread can also call this function directly after extra initialization.

Different kthreads can be used for the same `kthread_worker` as long as there's only one kthread attached to it at any given time. A `kthread_worker` without an attached kthread simply collects queued `kthread_works`.

## Name

`queue_kthread_work` — queue a `kthread_work`

## Synopsis

```
bool queue_kthread_work (struct kthread_worker * worker, struct
kthread_work * work);
```

## Arguments

*worker* target `kthread_worker`

*work* `kthread_work` to queue

## Description

Queue *work* to work processor *task* for async execution. *task* must have been created with `kthread_worker_create`. Returns `true` if *work* was successfully queued, `false` if it was already pending.

## Name

`flush_kthread_work` — flush a `kthread_work`

## Synopsis

```
void flush_kthread_work (struct kthread_work * work);
```

## Arguments

*work*    work to flush

## Description

If *work* is queued or executing, wait for it to finish execution.

## Name

`flush_kthread_worker` — flush all current works on a `kthread_worker`

## Synopsis

```
void flush_kthread_worker (struct kthread_worker * worker);
```

## Arguments

*worker*   worker to flush

## Description

Wait until all currently executing or pending works on *worker* are finished.

# Kernel objects manipulation

## Name

`kobject_get_path` — generate and return the path associated with a given `kobj` and `kset` pair.

## Synopsis

```
char * kobject_get_path (struct kobject * kobj, gfp_t gfp_mask);
```

## Arguments

*kobj*            `kobject` in question, with which to build the path

*gfp\_mask*      the allocation type used to allocate the path

## Description

The result must be freed by the caller with `kfree`.



## Name

`kobject_set_name` — Set the name of a `kobject`

## Synopsis

```
int kobject_set_name (struct kobject * kobj, const char * fmt, ...);
```

## Arguments

*kobj*    struct `kobject` to set the name of

*fmt*    format string used to build the name

*...*    variable arguments

## Description

This sets the name of the `kobject`. If you have already added the `kobject` to the system, you must call `kobject_rename` in order to change the name of the `kobject`.

## Name

`kobject_init` — initialize a kobject structure

## Synopsis

```
void kobject_init (struct kobject * kobj, struct kobj_type * ktype);
```

## Arguments

*kobj*     pointer to the kobject to initialize

*ktype*   pointer to the ktype for this kobject.

## Description

This function will properly initialize a kobject such that it can then be passed to the `kobject_add` call.

After this function is called, the kobject **MUST** be cleaned up by a call to `kobject_put`, not by a call to `kfree` directly to ensure that all of the memory is cleaned up properly.

## Name

`kobject_add` — the main kobject add function

## Synopsis

```
int kobject_add (struct kobject * kobj, struct kobject * parent, const
char * fmt, ...);
```

## Arguments

<i>kobj</i>	the kobject to add
<i>parent</i>	pointer to the parent of the kobject.
<i>fmt</i>	format to name the kobject with.
...	variable arguments

## Description

The kobject name is set and added to the kobject hierarchy in this function.

If *parent* is set, then the parent of the *kobj* will be set to it. If *parent* is NULL, then the parent of the *kobj* will be set to the kobject associated with the kset assigned to this kobject. If no kset is assigned to the kobject, then the kobject will be located in the root of the sysfs tree.

If this function returns an error, `kobject_put` must be called to properly clean up the memory associated with the object. Under no instance should the kobject that is passed to this function be directly freed with a call to `kfree`, that can leak memory.

Note, no “add” uevent will be created with this call, the caller should set up all of the necessary sysfs files for the object and then call `kobject_uevent` with the `UEVENT_ADD` parameter to ensure that userspace is properly notified of this kobject's creation.

## Name

`kobject_init_and_add` — initialize a kobject structure and add it to the kobject hierarchy

## Synopsis

```
int kobject_init_and_add (struct kobject * kobj, struct kobj_type *  
ktype, struct kobject * parent, const char * fmt, ...);
```

## Arguments

<i>kobj</i>	pointer to the kobject to initialize
<i>ktype</i>	pointer to the ktype for this kobject.
<i>parent</i>	pointer to the parent of this kobject.
<i>fmt</i>	the name of the kobject.
...	variable arguments

## Description

This function combines the call to `kobject_init` and `kobject_add`. The same type of error handling after a call to `kobject_add` and kobject lifetime rules are the same here.

## Name

`kobject_rename` — change the name of an object

## Synopsis

```
int kobject_rename (struct kobject * kobj, const char * new_name);
```

## Arguments

*kobj*            object in question.

*new\_name*      object's new name

## Description

It is the responsibility of the caller to provide mutual exclusion between two different calls of `kobject_rename` on the same `kobject` and to ensure that `new_name` is valid and won't conflict with other `kobjects`.

## Name

`kobject_move` — move object to another parent

## Synopsis

```
int kobject_move (struct kobject * kobj, struct kobject * new_parent);
```

## Arguments

*kobj*                object in question.

*new\_parent*        object's new parent (can be NULL)

## Name

`kobject_del` — unlink `kobject` from hierarchy.

## Synopsis

```
void kobject_del (struct kobject * kobj);
```

## Arguments

*kobj* object.

## Name

`kobject_get` — increment refcount for object.

## Synopsis

```
struct kobject * kobject_get (struct kobject * kobj);
```

## Arguments

*kobj* object.



## Name

`kobject_put` — decrement refcount for object.

## Synopsis

```
void kobject_put (struct kobject * kobj);
```

## Arguments

*kobj* object.

## Description

Decrement the refcount, and if 0, call `kobject_cleanup`.

## Name

`kobject_create_and_add` — create a struct `kobject` dynamically and register it with `sysfs`

## Synopsis

```
struct kobject * kobject_create_and_add (const char * name, struct
kobject * parent);
```

## Arguments

*name*      the name for the `kobject`

*parent*    the parent `kobject` of this `kobject`, if any.

## Description

This function creates a `kobject` structure dynamically and registers it with `sysfs`. When you are finished with this structure, call `kobject_put` and the structure will be dynamically freed when it is no longer being used.

If the `kobject` was not able to be created, `NULL` will be returned.

## Name

`kset_register` — initialize and add a kset.

## Synopsis

```
int kset_register (struct kset * k);
```

## Arguments

*k* kset.

## Name

`kset_unregister` — remove a kset.

## Synopsis

```
void kset_unregister (struct kset * k);
```

## Arguments

*k* kset.

## Name

`kset_create_and_add` — create a struct `kset` dynamically and add it to `sysfs`

## Synopsis

```
struct kset * kset_create_and_add (const char * name, const struct  
kset_uevent_ops * uevent_ops, struct kobject * parent_kobj);
```

## Arguments

*name*                    the name for the `kset`

*uevent\_ops*            a struct `kset_uevent_ops` for the `kset`

*parent\_kobj*           the parent `kobject` of this `kset`, if any.

## Description

This function creates a `kset` structure dynamically and registers it with `sysfs`. When you are finished with this structure, call `kset_unregister` and the structure will be dynamically freed when it is no longer being used.

If the `kset` was not able to be created, `NULL` will be returned.

## Kernel utility functions

## Name

`upper_32_bits` — return bits 32-63 of a number

## Synopsis

```
upper_32_bits ( n );
```

## Arguments

*n* the number we're accessing

## Description

A basic shift-right of a 64- or 32-bit quantity. Use this to suppress the “right shift count  $\geq$  width of type” warning when that quantity is 32-bits.

## Name

`lower_32_bits` — return bits 0-31 of a number

## Synopsis

```
lower_32_bits ( n );
```

## Arguments

*n* the number we're accessing

## Name

`might_sleep` — annotation for functions that can sleep

## Synopsis

```
might_sleep (void);
```

## Arguments

None

## Description

this macro will print a stack trace if it is executed in an atomic context (spinlock, irq-handler, ...).

This is a useful debugging help to be able to catch problems early and not be bitten later when the calling function happens to sleep when it is not supposed to.



## Name

`abs` — return absolute value of an argument

## Synopsis

```
abs ( x );
```

## Arguments

*x* the value. If it is unsigned type, it is converted to signed type first (s64, long or int depending on its size).

## Return

an absolute value of *x*. If *x* is 64-bit, macro's return type is s64, otherwise it is signed long.

## Name

`reciprocal_scale` — "scale" a value into range  $[0, ep\_ro)$

## Synopsis

```
u32 reciprocal_scale (u32 val, u32 ep_ro);
```

## Arguments

*val*      value

*ep\_ro*    right open interval endpoint

## Description

Perform a “reciprocal multiplication” in order to “scale” a value into range  $[0, ep\_ro)$ , where the upper interval endpoint is right-open. This is useful, e.g. for accessing a index of an array containing `ep_ro` elements, for example. Think of it as sort of modulus, only that the result isn't that of modulo. ;) Note that if initial input is a small value, then result will return 0.

## Return

a result based on `val` in interval  $[0, ep\_ro)$ .

## 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

kstrtol — convert a string to a long

## Synopsis

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

`trace_printk` — printf formatting in the ftrace buffer

## Synopsis

```
trace_printk ( fmt, ... );
```

## Arguments

*fmt*    the printf format for printing

...    variable arguments

## Note

`__trace_printk` is an internal function for `trace_printk` and the *ip* is passed in via the `trace_printk` macro.

This function allows a kernel developer to debug fast path sections that `printk` is not appropriate for. By scattering in various `printk` like tracing in the code, a developer can quickly see where problems are occurring.

This is intended as a debugging tool for the developer only. Please refrain from leaving `trace_printks` scattered around in your code. (Extra memory is used for special buffers that are allocated when `trace_printk` is used)

A little optimization trick is done here. If there's only one argument, there's no need to scan the string for printf formats. The `trace_puts` will suffice. But how can we take advantage of using `trace_puts` when `trace_printk` has only one argument? By stringifying the args and checking the size we can tell whether or not there are args. `__stringify((__VA_ARGS__))` will turn into `"()\0"` with a size of 3 when there are no args, anything else will be bigger. All we need to do is define a string to this, and then take its size and compare to 3. If it's bigger, use `do_trace_printk` otherwise, optimize it to `trace_puts`. Then just let gcc optimize the rest.

## Name

`trace_puts` — write a string into the ftrace buffer

## Synopsis

```
trace_puts ( str );
```

## Arguments

*str* the string to record

## Note

`__trace_bputs` is an internal function for `trace_puts` and the *ip* is passed in via the `trace_puts` macro.

This is similar to `trace_printk` but is made for those really fast paths that a developer wants the least amount of “Heisenbug” affects, where the processing of the print format is still too much.

This function allows a kernel developer to debug fast path sections that `printk` is not appropriate for. By scattering in various `printk` like tracing in the code, a developer can quickly see where problems are occurring.

This is intended as a debugging tool for the developer only. Please refrain from leaving `trace_puts` scattered around in your code. (Extra memory is used for special buffers that are allocated when `trace_puts` is used)

## Returns

0 if nothing was written, positive # if string was. (1 when `__trace_bputs` is used, `strlen(str)` when `__trace_puts` is used)

## Name

`min_not_zero` — return the minimum that is `_not_ zero`, unless both are zero

## Synopsis

```
min_not_zero ( x, y );
```

## Arguments

*x* value1

*y* value2

## Name

`clamp` — return a value clamped to a given range with strict typechecking

## Synopsis

```
clamp ( val, lo, hi );
```

## Arguments

*val*    current value

*lo*    lowest allowable value

*hi*    highest allowable value

## Description

This macro does strict typechecking of *lo*/*hi* to make sure they are of the same type as *val*. See the unnecessary pointer comparisons.



## Name

`clamp_t` — return a value clamped to a given range using a given type

## Synopsis

```
clamp_t ( type, val, lo, hi );
```

## Arguments

*type*    the type of variable to use

*val*     current value

*lo*      minimum allowable value

*hi*      maximum allowable value

## Description

This macro does no typechecking and uses temporary variables of type 'type' to make all the comparisons.

## Name

`clamp_val` — return a value clamped to a given range using `val`'s type

## Synopsis

```
clamp_val ( val, lo, hi );
```

## Arguments

*val*    current value

*lo*    minimum allowable value

*hi*    maximum allowable value

## Description

This macro does no typechecking and uses temporary variables of whatever type the input argument '`val`' is. This is useful when `val` is an unsigned type and `min` and `max` are literals that will otherwise be assigned a signed integer type.

## Name

`container_of` — cast a member of a structure out to the containing structure

## Synopsis

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

## Arguments

*ptr*        the pointer to the member.

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

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

## Name

`printk` — print a kernel message

## Synopsis

```
__visible int printk (const char * fmt, ...);
```

## Arguments

*fmt*    format string

...    variable arguments

## Description

This is `printk`. It can be called from any context. We want it to work.

We try to grab the `console_lock`. If we succeed, it's easy - we log the output and call the console drivers. If we fail to get the semaphore, we place the output into the log buffer and return. The current holder of the `console_sem` will notice the new output in `console_unlock`; and will send it to the consoles before releasing the lock.

One effect of this deferred printing is that code which calls `printk` and then changes `console_loglevel` may break. This is because `console_loglevel` is inspected when the actual printing occurs.

## See also

`printf(3)`

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

## Name

`console_lock` — lock the console system for exclusive use.

## Synopsis

```
void console_lock ( void );
```

## Arguments

*void* no arguments

## Description

Acquires a lock which guarantees that the caller has exclusive access to the console system and the `console_drivers` list.

Can sleep, returns nothing.

## Name

`console_trylock` — try to lock the console system for exclusive use.

## Synopsis

```
int console_trylock ( void );
```

## Arguments

*void* no arguments

## Description

Try to acquire a lock which guarantees that the caller has exclusive access to the console system and the `console_drivers` list.

returns 1 on success, and 0 on failure to acquire the lock.

## Name

`console_unlock` — unlock the console system

## Synopsis

```
void console_unlock ( void );
```

## Arguments

*void* no arguments

## Description

Releases the `console_lock` which the caller holds on the console system and the console driver list.

While the `console_lock` was held, console output may have been buffered by `printk`. If this is the case, `console_unlock`; emits the output prior to releasing the lock.

If there is output waiting, we wake `/dev/kmsg` and `syslog` users.

`console_unlock`; may be called from any context.

## Name

`console_conditional_schedule` — yield the CPU if required

## Synopsis

```
void __sched console_conditional_schedule ( void );
```

## Arguments

*void* no arguments

## Description

If the console code is currently allowed to sleep, and if this CPU should yield the CPU to another task, do so here.

Must be called within `console_lock`;



## Name

`printk_timed_ratelimit` — caller-controlled printk ratelimiting

## Synopsis

```
bool printk_timed_ratelimit (unsigned long * caller_jiffies, unsigned
int interval_msecs);
```

## Arguments

*caller\_jiffies*   pointer to caller's state

*interval\_msecs*   minimum interval between prints

## Description

`printk_timed_ratelimit` returns true if more than *interval\_msecs* milliseconds have elapsed since the last time `printk_timed_ratelimit` returned true.

## Name

`kmsg_dump_register` — register a kernel log dumper.

## Synopsis

```
int kmsg_dump_register (struct kmsg_dumper * dumper);
```

## Arguments

*dumper* pointer to the `kmsg_dumper` structure

## Description

Adds a kernel log dumper to the system. The dump callback in the structure will be called when the kernel oopses or panics and must be set. Returns zero on success and `-EINVAL` or `-EBUSY` otherwise.

## Name

`kmsg_dump_unregister` — unregister a kmsg dumper.

## Synopsis

```
int kmsg_dump_unregister (struct kmsg_dumper * dumper);
```

## Arguments

*dumper* pointer to the kmsg\_dumper structure

## Description

Removes a dump device from the system. Returns zero on success and `-EINVAL` otherwise.

## Name

`kmsg_dump_get_line` — retrieve one kmsg log line

## Synopsis

```
bool kmsg_dump_get_line (struct kmsg_dumper * dumper, bool syslog, char  
* line, size_t size, size_t * len);
```

## Arguments

*dumper*    registered kmsg dumper

*syslog*    include the “<4>” prefixes

*line*       buffer to copy the line to

*size*       maximum size of the buffer

*len*        length of line placed into buffer

## Description

Start at the beginning of the kmsg buffer, with the oldest kmsg record, and copy one record into the provided buffer.

Consecutive calls will return the next available record moving towards the end of the buffer with the youngest messages.

A return value of FALSE indicates that there are no more records to read.

## Name

`kmsg_dump_get_buffer` — copy kmsg log lines

## Synopsis

```
bool kmsg_dump_get_buffer (struct kmsg_dumper * dumper, bool syslog,  
char * buf, size_t size, size_t * len);
```

## Arguments

*dumper*    registered kmsg dumper

*syslog*    include the “<4>” prefixes

*buf*        buffer to copy the line to

*size*        maximum size of the buffer

*len*        length of line placed into buffer

## Description

Start at the end of the kmsg buffer and fill the provided buffer with as many of the the \*youngest\* kmsg records that fit into it. If the buffer is large enough, all available kmsg records will be copied with a single call.

Consecutive calls will fill the buffer with the next block of available older records, not including the earlier retrieved ones.

A return value of FALSE indicates that there are no more records to read.

## Name

`kmsg_dump_rewind` — reset the iterator

## Synopsis

```
void kmsg_dump_rewind (struct kmsg_dumper * dumper);
```

## Arguments

*dumper* registered kmsg dumper

## Description

Reset the dumper's iterator so that `kmsg_dump_get_line` and `kmsg_dump_get_buffer` can be called again and used multiple times within the same `dumper.dump` callback.

## Name

`printk_hash` — print a kernel message include a hash over the message

## Synopsis

```
int printk_hash (const char * prefix, const char * fmt, ...);
```

## Arguments

*prefix*    message prefix including the “.06x” for the hash

*fmt*        format string

*...*        variable arguments

## Name

`printk_dev_hash` — print a kernel message include a hash over the message

## Synopsis

```
int printk_dev_hash (const char * prefix, const char * driver_name,  
const char * fmt, ...);
```

## Arguments

*prefix*            message prefix including the “.06x” for the hash

*driver\_name*    -- undescribed --

*fmt*             format string

*...*            variable arguments



## Name

panic — halt the system

## Synopsis

```
void panic (const char * fmt, ...);
```

## Arguments

*fmt*    The text string to print

...    variable arguments

## Description

Display a message, then perform cleanups.

This function never returns.

## Name

`add_taint` —

## Synopsis

```
void add_taint (unsigned flag, enum lockdep_ok lockdep_ok);
```

## Arguments

*flag*                    one of the `TAINT_*` constants.

*lockdep\_ok*    whether lock debugging is still OK.

## Description

If something bad has gone wrong, you'll want `lockdebug_ok` = false, but for some noteworthy-but-not-corrupting cases, it can be set to true.

## Name

kernel/sys.c — Document generation inconsistency

## Oops

### Warning

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

## Name

`init_srcu_struct` — initialize a sleep-RCU structure

## Synopsis

```
int init_srcu_struct (struct srcu_struct * sp);
```

## Arguments

*sp* structure to initialize.

## Description

Must invoke this on a given `srcu_struct` before passing that `srcu_struct` to any other function. Each `srcu_struct` represents a separate domain of SRCU protection.

## Name

`cleanup_srcu_struct` — deconstruct a sleep-RCU structure

## Synopsis

```
void cleanup_srcu_struct (struct srcu_struct * sp);
```

## Arguments

*sp*    structure to clean up.

## Description

Must invoke this after you are finished using a given `srcu_struct` that was initialized via `init_srcu_struct`, else you leak memory.

## Name

`synchronize_srcu` — wait for prior SRCU read-side critical-section completion

## Synopsis

```
void synchronize_srcu (struct srcu_struct * sp);
```

## Arguments

*sp* srcu\_struct with which to synchronize.

## Description

Wait for the count to drain to zero of both indexes. To avoid the possible starvation of `synchronize_srcu`, it waits for the count of the index= $((\rightarrow\text{completed} \& 1) \wedge 1)$  to drain to zero at first, and then flip the completed and wait for the count of the other index.

Can block; must be called from process context.

Note that it is illegal to call `synchronize_srcu` from the corresponding SRCU read-side critical section; doing so will result in deadlock. However, it is perfectly legal to call `synchronize_srcu` on one `srcu_struct` from some other `srcu_struct`'s read-side critical section, as long as the resulting graph of `srcu_structs` is acyclic.

There are memory-ordering constraints implied by `synchronize_srcu`. On systems with more than one CPU, when `synchronize_srcu` returns, each CPU is guaranteed to have executed a full memory barrier since the end of its last corresponding SRCU-sched read-side critical section whose beginning preceded the call to `synchronize_srcu`. In addition, each CPU having an SRCU read-side critical section that extends beyond the return from `synchronize_srcu` is guaranteed to have executed a full memory barrier after the beginning of `synchronize_srcu` and before the beginning of that SRCU read-side critical section. Note that these guarantees include CPUs that are offline, idle, or executing in user mode, as well as CPUs that are executing in the kernel.

Furthermore, if CPU A invoked `synchronize_srcu`, which returned to its caller on CPU B, then both CPU A and CPU B are guaranteed to have executed a full memory barrier during the execution of `synchronize_srcu`. This guarantee applies even if CPU A and CPU B are the same CPU, but again only if the system has more than one CPU.

Of course, these memory-ordering guarantees apply only when `synchronize_srcu`, `srcu_read_lock`, and `srcu_read_unlock` are passed the same `srcu_struct` structure.

## Name

`synchronize_srcu_expedited` — Brute-force SRCU grace period

## Synopsis

```
void synchronize_srcu_expedited (struct srcu_struct * sp);
```

## Arguments

*sp*    `srcu_struct` with which to synchronize.

## Description

Wait for an SRCU grace period to elapse, but be more aggressive about spinning rather than blocking when waiting.

Note that `synchronize_srcu_expedited` has the same deadlock and memory-ordering properties as does `synchronize_srcu`.

## Name

`srcu_barrier` — Wait until all in-flight `call_srcu` callbacks complete.

## Synopsis

```
void srcu_barrier (struct srcu_struct * sp);
```

## Arguments

*sp*   `srcu_struct` on which to wait for in-flight callbacks.



## Name

`srcu_batches_completed` — return batches completed.

## Synopsis

```
unsigned long srcu_batches_completed (struct srcu_struct * sp);
```

## Arguments

*sp*    `srcu_struct` on which to report batch completion.

## Description

Report the number of batches, correlated with, but not necessarily precisely the same as, the number of grace periods that have elapsed.

## Name

`rcu_idle_enter` — inform RCU that current CPU is entering idle

## Synopsis

```
void rcu_idle_enter ( void );
```

## Arguments

*void* no arguments

## Description

Enter idle mode, in other words, -leave- the mode in which RCU read-side critical sections can occur. (Though RCU read-side critical sections can occur in irq handlers in idle, a possibility handled by `irq_enter` and `irq_exit`.)

We crowbar the `->dynticks_nesting` field to zero to allow for the possibility of usermode upcalls having messed up our count of interrupt nesting level during the prior busy period.

## Name

`rcu_idle_exit` — inform RCU that current CPU is leaving idle

## Synopsis

```
void rcu_idle_exit ( void );
```

## Arguments

*void* no arguments

## Description

Exit idle mode, in other words, -enter- the mode in which RCU read-side critical sections can occur.

We crowbar the `->dynticks_nesting` field to `DYNTICK_TASK_NEST` to allow for the possibility of usermode upcalls messing up our count of interrupt nesting level during the busy period that is just now starting.

## Name

`rcu_is_watching` — see if RCU thinks that the current CPU is idle

## Synopsis

```
bool notrace rcu_is_watching ( void );
```

## Arguments

*void* no arguments

## Description

If the current CPU is in its idle loop and is neither in an interrupt or NMI handler, return true.

## Name

`synchronize_sched` — wait until an rcu-sched grace period has elapsed.

## Synopsis

```
void synchronize_sched ( void );
```

## Arguments

*void* no arguments

## Description

Control will return to the caller some time after a full rcu-sched grace period has elapsed, in other words after all currently executing rcu-sched read-side critical sections have completed. These read-side critical sections are delimited by `rcu_read_lock_sched` and `rcu_read_unlock_sched`, and may be nested. Note that `preempt_disable`, `local_irq_disable`, and so on may be used in place of `rcu_read_lock_sched`.

This means that all `preempt_disable` code sequences, including NMI and non-threaded hardware-interrupt handlers, in progress on entry will have completed before this primitive returns. However, this does not guarantee that softirq handlers will have completed, since in some kernels, these handlers can run in process context, and can block.

Note that this guarantee implies further memory-ordering guarantees. On systems with more than one CPU, when `synchronize_sched` returns, each CPU is guaranteed to have executed a full memory barrier since the end of its last RCU-sched read-side critical section whose beginning preceded the call to `synchronize_sched`. In addition, each CPU having an RCU read-side critical section that extends beyond the return from `synchronize_sched` is guaranteed to have executed a full memory barrier after the beginning of `synchronize_sched` and before the beginning of that RCU read-side critical section. Note that these guarantees include CPUs that are offline, idle, or executing in user mode, as well as CPUs that are executing in the kernel.

Furthermore, if CPU A invoked `synchronize_sched`, which returned to its caller on CPU B, then both CPU A and CPU B are guaranteed to have executed a full memory barrier during the execution of `synchronize_sched` -- even if CPU A and CPU B are the same CPU (but again only if the system has more than one CPU).

This primitive provides the guarantees made by the (now removed) `synchronize_kernel` API. In contrast, `synchronize_rcu` only guarantees that `rcu_read_lock` sections will have completed. In “classic RCU”, these two guarantees happen to be one and the same, but can differ in realtime RCU implementations.

## Name

`synchronize_rcu_bh` — wait until an `rcu_bh` grace period has elapsed.

## Synopsis

```
void synchronize_rcu_bh ( void );
```

## Arguments

*void* no arguments

## Description

Control will return to the caller some time after a full `rcu_bh` grace period has elapsed, in other words after all currently executing `rcu_bh` read-side critical sections have completed. RCU read-side critical sections are delimited by `rcu_read_lock_bh` and `rcu_read_unlock_bh`, and may be nested.

See the description of `synchronize_sched` for more detailed information on memory ordering guarantees.

## Name

`get_state_synchronize_rcu` — Snapshot current RCU state

## Synopsis

```
unsigned long get_state_synchronize_rcu ( void );
```

## Arguments

*void* no arguments

## Description

Returns a cookie that is used by a later call to `cond_synchronize_rcu` to determine whether or not a full grace period has elapsed in the meantime.

## Name

`cond_synchronize_rcu` — Conditionally wait for an RCU grace period

## Synopsis

```
void cond_synchronize_rcu (unsigned long oldstate);
```

## Arguments

*oldstate*    return value from earlier call to `get_state_synchronize_rcu`

## Description

If a full RCU grace period has elapsed since the earlier call to `get_state_synchronize_rcu`, just return. Otherwise, invoke `synchronize_rcu` to wait for a full grace period.

Yes, this function does not take counter wrap into account. But counter wrap is harmless. If the counter wraps, we have waited for more than 2 billion grace periods (and way more on a 64-bit system!), so waiting for one additional grace period should be just fine.



## Name

`get_state_synchronize_sched` — Snapshot current RCU-sched state

## Synopsis

```
unsigned long get_state_synchronize_sched ( void );
```

## Arguments

*void* no arguments

## Description

Returns a cookie that is used by a later call to `cond_synchronize_sched` to determine whether or not a full grace period has elapsed in the meantime.

## Name

`cond_synchronize_sched` — Conditionally wait for an RCU-sched grace period

## Synopsis

```
void cond_synchronize_sched (unsigned long oldstate);
```

## Arguments

*oldstate*    return value from earlier call to `get_state_synchronize_sched`

## Description

If a full RCU-sched grace period has elapsed since the earlier call to `get_state_synchronize_sched`, just return. Otherwise, invoke `synchronize_sched` to wait for a full grace period.

Yes, this function does not take counter wrap into account. But counter wrap is harmless. If the counter wraps, we have waited for more than 2 billion grace periods (and way more on a 64-bit system!), so waiting for one additional grace period should be just fine.

## Name

`synchronize_sched_expedited` — Brute-force RCU-sched grace period

## Synopsis

```
void synchronize_sched_expedited ( void );
```

## Arguments

*void* no arguments

## Description

Wait for an RCU-sched grace period to elapse, but use a “big hammer” approach to force the grace period to end quickly. This consumes significant time on all CPUs and is unfriendly to real-time workloads, so is thus not recommended for any sort of common-case code. In fact, if you are using `synchronize_sched_expedited` in a loop, please restructure your code to batch your updates, and then use a single `synchronize_sched` instead.

This implementation can be thought of as an application of sequence locking to expedited grace periods, but using the sequence counter to determine when someone else has already done the work instead of for retrying readers.

## Name

`rcu_barrier_bh` — Wait until all in-flight `call_rcu_bh` callbacks complete.

## Synopsis

```
void rcu_barrier_bh ( void );
```

## Arguments

*void* no arguments

## Name

`rcu_barrier_sched` — Wait for in-flight `call_rcu_sched` callbacks.

## Synopsis

```
void rcu_barrier_sched ( void );
```

## Arguments

*void* no arguments

## Name

`synchronize_rcu` — wait until a grace period has elapsed.

## Synopsis

```
void synchronize_rcu ( void );
```

## Arguments

*void* no arguments

## Description

Control will return to the caller some time after a full grace period has elapsed, in other words after all currently executing RCU read-side critical sections have completed. Note, however, that upon return from `synchronize_rcu`, the caller might well be executing concurrently with new RCU read-side critical sections that began while `synchronize_rcu` was waiting. RCU read-side critical sections are delimited by `rcu_read_lock` and `rcu_read_unlock`, and may be nested.

See the description of `synchronize_sched` for more detailed information on memory ordering guarantees.

## Name

synchronize\_rcu\_expedited — Brute-force RCU grace period

## Synopsis

```
void synchronize_rcu_expedited ( void );
```

## Arguments

*void* no arguments

## Description

Wait for an RCU-preempt grace period, but expedite it. The basic idea is to invoke `synchronize_sched_expedited` to push all the tasks to the `->blkd_tasks` lists and wait for this list to drain. This consumes significant time on all CPUs and is unfriendly to real-time workloads, so is thus not recommended for any sort of common-case code. In fact, if you are using `synchronize_rcu_expedited` in a loop, please restructure your code to batch your updates, and then Use a single `synchronize_rcu` instead.

## Name

`rcu_barrier` — Wait until all in-flight `call_rcu` callbacks complete.

## Synopsis

```
void rcu_barrier ( void );
```

## Arguments

*void* no arguments

## Description

Note that this primitive does not necessarily wait for an RCU grace period to complete. For example, if there are no RCU callbacks queued anywhere in the system, then `rcu_barrier` is within its rights to return immediately, without waiting for anything, much less an RCU grace period.



## Name

`rcu_read_lock_sched_held` — might we be in RCU-sched read-side critical section?

## Synopsis

```
int rcu_read_lock_sched_held ( void );
```

## Arguments

*void* no arguments

## Description

If `CONFIG_DEBUG_LOCK_ALLOC` is selected, returns nonzero iff in an RCU-sched read-side critical section. In absence of `CONFIG_DEBUG_LOCK_ALLOC`, this assumes we are in an RCU-sched read-side critical section unless it can prove otherwise. Note that disabling of preemption (including disabling irqs) counts as an RCU-sched read-side critical section. This is useful for debug checks in functions that required that they be called within an RCU-sched read-side critical section.

Check `debug_lockdep_rcu_enabled` to prevent false positives during boot and while lockdep is disabled.

Note that if the CPU is in the idle loop from an RCU point of view (ie: that we are in the section between `rcu_idle_enter` and `rcu_idle_exit`) then `rcu_read_lock_held` returns false even if the CPU did an `rcu_read_lock`. The reason for this is that RCU ignores CPUs that are in such a section, considering these as in extended quiescent state, so such a CPU is effectively never in an RCU read-side critical section regardless of what RCU primitives it invokes. This state of affairs is required --- we need to keep an RCU-free window in idle where the CPU may possibly enter into low power mode. This way we can notice an extended quiescent state to other CPUs that started a grace period. Otherwise we would delay any grace period as long as we run in the idle task.

Similarly, we avoid claiming an SRCU read lock held if the current CPU is offline.

## Name

`rcu_expedite_gp` — Expedite future RCU grace periods

## Synopsis

```
void rcu_expedite_gp ( void );
```

## Arguments

*void* no arguments

## Description

After a call to this function, future calls to `synchronize_rcu` and friends act as the corresponding `synchronize_rcu_expedited` function had instead been called.

## Name

`rcu_unexpedite_gp` — Cancel prior `rcu_expedite_gp` invocation

## Synopsis

```
void rcu_unexpedite_gp ( void );
```

## Arguments

*void* no arguments

## Description

Undo a prior call to `rcu_expedite_gp`. If all prior calls to `rcu_expedite_gp` are undone by a subsequent call to `rcu_unexpedite_gp`, and if the `rcu_expedited` sysfs/boot parameter is not set, then all subsequent calls to `synchronize_rcu` and friends will return to their normal non-expedited behavior.

## Name

`rcu_read_lock_held` — might we be in RCU read-side critical section?

## Synopsis

```
int rcu_read_lock_held ( void );
```

## Arguments

*void* no arguments

## Description

If `CONFIG_DEBUG_LOCK_ALLOC` is selected, returns nonzero iff in an RCU read-side critical section. In absence of `CONFIG_DEBUG_LOCK_ALLOC`, this assumes we are in an RCU read-side critical section unless it can prove otherwise. This is useful for debug checks in functions that require that they be called within an RCU read-side critical section.

Checks `debug_lockdep_rcu_enabled` to prevent false positives during boot and while lockdep is disabled.

Note that `rcu_read_lock` and the matching `rcu_read_unlock` must occur in the same context, for example, it is illegal to invoke `rcu_read_unlock` in process context if the matching `rcu_read_lock` was invoked from within an irq handler.

Note that `rcu_read_lock` is disallowed if the CPU is either idle or offline from an RCU perspective, so check for those as well.

## Name

`rcu_read_lock_bh_held` — might we be in RCU-bh read-side critical section?

## Synopsis

```
int rcu_read_lock_bh_held ( void );
```

## Arguments

*void* no arguments

## Description

Check for bottom half being disabled, which covers both the `CONFIG_PROVE_RCU` and not cases. Note that if someone uses `rcu_read_lock_bh`, but then later enables BH, lockdep (if enabled) will show the situation. This is useful for debug checks in functions that require that they be called within an RCU read-side critical section.

Check `debug_lockdep_rcu_enabled` to prevent false positives during boot.

Note that `rcu_read_lock` is disallowed if the CPU is either idle or offline from an RCU perspective, so check for those as well.

## Name

wakeme\_after\_rcu — Callback function to awaken a task after grace period

## Synopsis

```
void wakeme_after_rcu (struct rcu_head * head);
```

## Arguments

*head* Pointer to rcu\_head member within rcu\_synchronize structure

## Description

Awaken the corresponding task now that a grace period has elapsed.

## Name

`init_rcu_head_on_stack` — initialize on-stack `rcu_head` for debugobjects

## Synopsis

```
void init_rcu_head_on_stack (struct rcu_head * head);
```

## Arguments

*head* pointer to `rcu_head` structure to be initialized

## Description

This function informs debugobjects of a new `rcu_head` structure that has been allocated as an auto variable on the stack. This function is not required for `rcu_head` structures that are statically defined or that are dynamically allocated on the heap. This function has no effect for !`CONFIG_DEBUG_OBJECTS_RCU_HEAD` kernel builds.

## Name

`destroy_rcu_head_on_stack` — destroy on-stack `rcu_head` for debugobjects

## Synopsis

```
void destroy_rcu_head_on_stack (struct rcu_head * head);
```

## Arguments

*head* pointer to `rcu_head` structure to be initialized

## Description

This function informs debugobjects that an on-stack `rcu_head` structure is about to go out of scope. As with `init_rcu_head_on_stack`, this function is not required for `rcu_head` structures that are statically defined or that are dynamically allocated on the heap. Also as with `init_rcu_head_on_stack`, this function has no effect for `!CONFIG_DEBUG_OBJECTS_RCU_HEAD` kernel builds.



## Name

`synchronize_rcu_tasks` — wait until an rcu-tasks grace period has elapsed.

## Synopsis

```
void synchronize_rcu_tasks ( void );
```

## Arguments

*void* no arguments

## Description

Control will return to the caller some time after a full rcu-tasks grace period has elapsed, in other words after all currently executing rcu-tasks read-side critical sections have elapsed. These read-side critical sections are delimited by calls to `schedule`, `cond_resched_rcu_qs`, idle execution, userspace execution, calls to `synchronize_rcu_tasks`, and (in theory, anyway) `cond_resched`.

This is a very specialized primitive, intended only for a few uses in tracing and other situations requiring manipulation of function preambles and profiling hooks. The `synchronize_rcu_tasks` function is not (yet) intended for heavy use from multiple CPUs.

Note that this guarantee implies further memory-ordering guarantees. On systems with more than one CPU, when `synchronize_rcu_tasks` returns, each CPU is guaranteed to have executed a full memory barrier since the end of its last RCU-tasks read-side critical section whose beginning preceded the call to `synchronize_rcu_tasks`. In addition, each CPU having an RCU-tasks read-side critical section that extends beyond the return from `synchronize_rcu_tasks` is guaranteed to have executed a full memory barrier after the beginning of `synchronize_rcu_tasks` and before the beginning of that RCU-tasks read-side critical section. Note that these guarantees include CPUs that are offline, idle, or executing in user mode, as well as CPUs that are executing in the kernel.

Furthermore, if CPU A invoked `synchronize_rcu_tasks`, which returned to its caller on CPU B, then both CPU A and CPU B are guaranteed to have executed a full memory barrier during the execution of `synchronize_rcu_tasks` -- even if CPU A and CPU B are the same CPU (but again only if the system has more than one CPU).

## Name

`rcu_barrier_tasks` — Wait for in-flight `call_rcu_tasks` callbacks.

## Synopsis

```
void rcu_barrier_tasks ( void );
```

## Arguments

*void* no arguments

## Description

Although the current implementation is guaranteed to wait, it is not obligated to, for example, if there are no pending callbacks.

# Device Resource Management

## Name

`devres_alloc_node` — Allocate device resource data

## Synopsis

```
void * devres_alloc_node (dr_release_t release, size_t size, gfp_t gfp,  
int nid);
```

## Arguments

*release*    Release function devres will be associated with

*size*       Allocation size

*gfp*        Allocation flags

*nid*        NUMA node

## Description

Allocate devres of *size* bytes. The allocated area is zeroed, then associated with *release*. The returned pointer can be passed to other `devres_*`() functions.

## RETURNS

Pointer to allocated devres on success, NULL on failure.

## Name

devres\_for\_each\_res — Resource iterator

## Synopsis

```
void devres_for_each_res (struct device * dev, dr_release_t release,  
dr_match_t match, void * match_data, void (*fn) (struct device *, void  
*, void *), void * data);
```

## Arguments

<i>dev</i>	Device to iterate resource from
<i>release</i>	Look for resources associated with this release function
<i>match</i>	Match function (optional)
<i>match_data</i>	Data for the match function
<i>fn</i>	Function to be called for each matched resource.
<i>data</i>	Data for <i>fn</i> , the 3rd parameter of <i>fn</i>

## Description

Call *fn* for each devres of *dev* which is associated with *release* and for which *match* returns 1.

## RETURNS

void

## Name

`devres_free` — Free device resource data

## Synopsis

```
void devres_free (void * res);
```

## Arguments

*res*    Pointer to devres data to free

## Description

Free devres created with `devres_alloc`.

## Name

`devres_add` — Register device resource

## Synopsis

```
void devres_add (struct device * dev, void * res);
```

## Arguments

*dev*    Device to add resource to

*res*    Resource to register

## Description

Register devres *res* to *dev*. *res* should have been allocated using `devres_alloc`. On driver detach, the associated release function will be invoked and devres will be freed automatically.

## Name

`devres_find` — Find device resource

## Synopsis

```
void * devres_find (struct device * dev, dr_release_t release, dr_match_t  
match, void * match_data);
```

## Arguments

<i>dev</i>	Device to lookup resource from
<i>release</i>	Look for resources associated with this release function
<i>match</i>	Match function (optional)
<i>match_data</i>	Data for the match function

## Description

Find the latest devres of *dev* which is associated with *release* and for which *match* returns 1. If *match* is NULL, it's considered to match all.

## RETURNS

Pointer to found devres, NULL if not found.

## Name

`devres_get` — Find devres, if non-existent, add one atomically

## Synopsis

```
void * devres_get (struct device * dev, void * new_res, dr_match_t  
match, void * match_data);
```

## Arguments

<i>dev</i>	Device to lookup or add devres for
<i>new_res</i>	Pointer to new initialized devres to add if not found
<i>match</i>	Match function (optional)
<i>match_data</i>	Data for the match function

## Description

Find the latest devres of *dev* which has the same release function as *new\_res* and for which *match* return 1. If found, *new\_res* is freed; otherwise, *new\_res* is added atomically.

## RETURNS

Pointer to found or added devres.



## Name

`devres_remove` — Find a device resource and remove it

## Synopsis

```
void * devres_remove (struct device * dev, dr_release_t release,  
dr_match_t match, void * match_data);
```

## Arguments

<i>dev</i>	Device to find resource from
<i>release</i>	Look for resources associated with this release function
<i>match</i>	Match function (optional)
<i>match_data</i>	Data for the match function

## Description

Find the latest devres of *dev* associated with *release* and for which *match* returns 1. If *match* is NULL, it's considered to match all. If found, the resource is removed atomically and returned.

## RETURNS

Pointer to removed devres on success, NULL if not found.

## Name

`devres_destroy` — Find a device resource and destroy it

## Synopsis

```
int devres_destroy (struct device * dev, dr_release_t release,
dr_match_t match, void * match_data);
```

## Arguments

<i>dev</i>	Device to find resource from
<i>release</i>	Look for resources associated with this release function
<i>match</i>	Match function (optional)
<i>match_data</i>	Data for the match function

## Description

Find the latest devres of *dev* associated with *release* and for which *match* returns 1. If *match* is NULL, it's considered to match all. If found, the resource is removed atomically and freed.

Note that the release function for the resource will not be called, only the devres-allocated data will be freed. The caller becomes responsible for freeing any other data.

## RETURNS

0 if devres is found and freed, -ENOENT if not found.

## Name

`devres_release` — Find a device resource and destroy it, calling `release`

## Synopsis

```
int devres_release (struct device * dev, dr_release_t release,
dr_match_t match, void * match_data);
```

## Arguments

<i>dev</i>	Device to find resource from
<i>release</i>	Look for resources associated with this release function
<i>match</i>	Match function (optional)
<i>match_data</i>	Data for the match function

## Description

Find the latest devres of *dev* associated with *release* and for which *match* returns 1. If *match* is NULL, it's considered to match all. If found, the resource is removed atomically, the release function called and the resource freed.

## RETURNS

0 if devres is found and freed, -ENOENT if not found.

## Name

`devres_open_group` — Open a new devres group

## Synopsis

```
void * devres_open_group (struct device * dev, void * id, gfp_t gfp);
```

## Arguments

*dev* Device to open devres group for

*id* Separator ID

*gfp* Allocation flags

## Description

Open a new devres group for *dev* with *id*. For *id*, using a pointer to an object which won't be used for another group is recommended. If *id* is NULL, address-wise unique ID is created.

## RETURNS

ID of the new group, NULL on failure.

## Name

`devres_close_group` — Close a devres group

## Synopsis

```
void devres_close_group (struct device * dev, void * id);
```

## Arguments

*dev*    Device to close devres group for

*id*     ID of target group, can be NULL

## Description

Close the group identified by *id*. If *id* is NULL, the latest open group is selected.

## Name

`devres_remove_group` — Remove a devres group

## Synopsis

```
void devres_remove_group (struct device * dev, void * id);
```

## Arguments

*dev*    Device to remove group for

*id*     ID of target group, can be NULL

## Description

Remove the group identified by *id*. If *id* is NULL, the latest open group is selected. Note that removing a group doesn't affect any other resources.

## Name

`devres_release_group` — Release resources in a devres group

## Synopsis

```
int devres_release_group (struct device * dev, void * id);
```

## Arguments

*dev*    Device to release group for

*id*     ID of target group, can be NULL

## Description

Release all resources in the group identified by *id*. If *id* is NULL, the latest open group is selected. The selected group and groups properly nested inside the selected group are removed.

## RETURNS

The number of released non-group resources.

## Name

`devm_add_action` — add a custom action to list of managed resources

## Synopsis

```
int devm_add_action (struct device * dev, void (*action) (void *), void  
* data);
```

## Arguments

*dev*        Device that owns the action

*action*    Function that should be called

*data*       Pointer to data passed to *action* implementation

## Description

This adds a custom action to the list of managed resources so that it gets executed as part of standard resource unwinding.



## Name

`devm_remove_action` — removes previously added custom action

## Synopsis

```
void devm_remove_action (struct device * dev, void (*action) (void *),  
void * data);
```

## Arguments

*dev*        Device that owns the action

*action*    Function implementing the action

*data*       Pointer to data passed to *action* implementation

## Description

Removes instance of *action* previously added by `devm_add_action`. Both action and data should match one of the existing entries.

## Name

`devm_kmalloc` — Resource-managed `kmalloc`

## Synopsis

```
void * devm_kmalloc (struct device * dev, size_t size, gfp_t gfp);
```

## Arguments

*dev*     Device to allocate memory for

*size*    Allocation size

*gfp*     Allocation gfp flags

## Description

Managed `kmalloc`. Memory allocated with this function is automatically freed on driver detach. Like all other devres resources, guaranteed alignment is unsigned long long.

## RETURNS

Pointer to allocated memory on success, NULL on failure.

## Name

`devm_kstrdup` — Allocate resource managed space and copy an existing string into that.

## Synopsis

```
char * devm_kstrdup (struct device * dev, const char * s, gfp_t gfp);
```

## Arguments

*dev* Device to allocate memory for

*s* the string to duplicate

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

## RETURNS

Pointer to allocated string on success, NULL on failure.

## Name

`devm_kvasprintf` — Allocate resource managed space and format a string into that.

## Synopsis

```
char * devm_kvasprintf (struct device * dev, gfp_t gfp, const char *  
fmt, va_list ap);
```

## Arguments

*dev* Device to allocate memory for

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

*fmt* The `printf`-style format string

*ap* Arguments for the format string

## RETURNS

Pointer to allocated string on success, NULL on failure.

## Name

`devm_kasprintf` — Allocate resource managed space and format a string into that.

## Synopsis

```
char * devm_kasprintf (struct device * dev, gfp_t gfp, const char *  
fmt, ...);
```

## Arguments

*dev* Device to allocate memory for

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

*fmt* The `printf`-style format string @...: Arguments for the format string

... variable arguments

## RETURNS

Pointer to allocated string on success, NULL on failure.

## Name

devm\_kfree — Resource-managed kfree

## Synopsis

```
void devm_kfree (struct device * dev, void * p);
```

## Arguments

*dev*    Device this memory belongs to

*p*      Memory to free

## Description

Free memory allocated with `devm_kmalloc`.

## Name

`devm_kmemdup` — Resource-managed `kmemdup`

## Synopsis

```
void * devm_kmemdup (struct device * dev, const void * src, size_t len,  
gfp_t gfp);
```

## Arguments

*dev* Device this memory belongs to

*src* Memory region to duplicate

*len* Memory region length

*gfp* GFP mask to use

## Description

Duplicate region of a memory using resource managed `kmalloc`

## Name

`devm_get_free_pages` — Resource-managed `__get_free_pages`

## Synopsis

```
unsigned long devm_get_free_pages (struct device * dev, gfp_t gfp_mask,  
unsigned int order);
```

## Arguments

<i>dev</i>	Device to allocate memory for
<i>gfp_mask</i>	Allocation gfp flags
<i>order</i>	Allocation size is $(1 \ll \text{order})$ pages

## Description

Managed `get_free_pages`. Memory allocated with this function is automatically freed on driver detach.

## RETURNS

Address of allocated memory on success, 0 on failure.



## Name

`devm_free_pages` — Resource-managed `free_pages`

## Synopsis

```
void devm_free_pages (struct device * dev, unsigned long addr);
```

## Arguments

*dev*     Device this memory belongs to

*addr*    Memory to free

## Description

Free memory allocated with `devm_get_free_pages`. Unlike `free_pages`, there is no need to supply the *order*.

---

# **Chapter 2. Device drivers infrastructure**

## **The Basic Device Driver-Model Structures**

## Name

struct bus\_type — The bus type of the device

## Synopsis

```
struct bus_type {
    const char * name;
    const char * dev_name;
    struct device * dev_root;
    struct device_attribute * dev_attrs;
    const struct attribute_group ** bus_groups;
    const struct attribute_group ** dev_groups;
    const struct attribute_group ** drv_groups;
    int (* match) (struct device *dev, struct device_driver *drv);
    int (* uevent) (struct device *dev, struct kobj_uevent_env *env);
    int (* probe) (struct device *dev);
    int (* remove) (struct device *dev);
    void (* shutdown) (struct device *dev);
    int (* online) (struct device *dev);
    int (* offline) (struct device *dev);
    int (* suspend) (struct device *dev, pm_message_t state);
    int (* resume) (struct device *dev);
    const struct dev_pm_ops * pm;
    const struct iommu_ops * iommu_ops;
    struct subsys_private * p;
    struct lock_class_key lock_key;
};
```

## Members

name	The name of the bus.
dev_name	Used for subsystems to enumerate devices like (“foou”, dev->id).
dev_root	Default device to use as the parent.
dev_attrs	Default attributes of the devices on the bus.
bus_groups	Default attributes of the bus.
dev_groups	Default attributes of the devices on the bus.
drv_groups	Default attributes of the device drivers on the bus.
match	Called, perhaps multiple times, whenever a new device or driver is added for this bus. It should return a nonzero value if the given device can be handled by the given driver.
uevent	Called when a device is added, removed, or a few other things that generate uevents to add the environment variables.
probe	Called when a new device or driver add to this bus, and callback the specific driver's probe to initial the matched device.
remove	Called when a device removed from this bus.

shutdown	Called at shut-down time to quiesce the device.
online	Called to put the device back online (after offlining it).
offline	Called to put the device offline for hot-removal. May fail.
suspend	Called when a device on this bus wants to go to sleep mode.
resume	Called to bring a device on this bus out of sleep mode.
pm	Power management operations of this bus, callback the specific device driver's pm-ops.
iommu_ops	IOMMU specific operations for this bus, used to attach IOMMU driver implementations to a bus and allow the driver to do bus-specific setup
p	The private data of the driver core, only the driver core can touch this.
lock_key	Lock class key for use by the lock validator

## Description

A bus is a channel between the processor and one or more devices. For the purposes of the device model, all devices are connected via a bus, even if it is an internal, virtual, “platform” bus. Buses can plug into each other. A USB controller is usually a PCI device, for example. The device model represents the actual connections between buses and the devices they control. A bus is represented by the `bus_type` structure. It contains the name, the default attributes, the bus' methods, PM operations, and the driver core's private data.

## Name

enum probe\_type — device driver probe type to try Device drivers may opt in for special handling of their respective probe routines. This tells the core what to expect and prefer.

## Synopsis

```
enum probe_type {  
    PROBE_DEFAULT_STRATEGY,  
    PROBE_PREFER_ASYNCHRONOUS,  
    PROBE_FORCE_SYNCHRONOUS  
};
```

## Constants

- |                           |  |
|---------------------------|--|
| PROBE_DEFAULT_STRATEGY    | Used by drivers that work equally well whether probed synchronously or asynchronously.   |
| PROBE_PREFER_ASYNCHRONOUS | Drivers for “slow” devices which probing order is not essential for booting the system may opt into executing their probes asynchronously.   |
| PROBE_FORCE_SYNCHRONOUS   | Use this to annotate drivers that need their probe routines to run synchronously with driver and device registration (with the exception of -EPROBE_DEFER handling - re-probing always ends up being done asynchronously). |

## Description

Note that the end goal is to switch the kernel to use asynchronous probing by default, so annotating drivers with PROBE\_PREFER\_ASYNCHRONOUS is a temporary measure that allows us to speed up boot process while we are validating the rest of the drivers.

## Name

struct device\_driver — The basic device driver structure

## Synopsis

```
struct device_driver {
    const char * name;
    struct bus_type * bus;
    struct module * owner;
    const char * mod_name;
    bool suppress_bind_attrs;
    enum probe_type probe_type;
    const struct of_device_id * of_match_table;
    const struct acpi_device_id * acpi_match_table;
    int (* probe) (struct device *dev);
    int (* remove) (struct device *dev);
    void (* shutdown) (struct device *dev);
    int (* suspend) (struct device *dev, pm_message_t state);
    int (* resume) (struct device *dev);
    const struct attribute_group ** groups;
    const struct dev_pm_ops * pm;
    struct driver_private * p;
};
```

## Members

name	Name of the device driver.
bus	The bus which the device of this driver belongs to.
owner	The module owner.
mod_name	Used for built-in modules.
suppress_bind_attrs	Disables bind/unbind via sysfs.
probe_type	Type of the probe (synchronous or asynchronous) to use.
of_match_table	The open firmware table.
acpi_match_table	The ACPI match table.
probe	Called to query the existence of a specific device, whether this driver can work with it, and bind the driver to a specific device.
remove	Called when the device is removed from the system to unbind a device from this driver.
shutdown	Called at shut-down time to quiesce the device.
suspend	Called to put the device to sleep mode. Usually to a low power state.
resume	Called to bring a device from sleep mode.
groups	Default attributes that get created by the driver core automatically.

pm	Power management operations of the device which matched this driver.
p	Driver core's private data, no one other than the driver core can touch this.

## Description

The device driver-model tracks all of the drivers known to the system. The main reason for this tracking is to enable the driver core to match up drivers with new devices. Once drivers are known objects within the system, however, a number of other things become possible. Device drivers can export information and configuration variables that are independent of any specific device.

## Name

struct subsys\_interface — interfaces to device functions

## Synopsis

```
struct subsys_interface {
    const char * name;
    struct bus_type * subsys;
    struct list_head node;
    int (* add_dev) (struct device *dev, struct subsys_interface *sif);
    void (* remove_dev) (struct device *dev, struct subsys_interface *sif);
};
```

## Members

name	name of the device function
subsys	subsystem of the devices to attach to
node	the list of functions registered at the subsystem
add_dev	device hookup to device function handler
remove_dev	device hookup to device function handler

## Description

Simple interfaces attached to a subsystem. Multiple interfaces can attach to a subsystem and its devices. Unlike drivers, they do not exclusively claim or control devices. Interfaces usually represent a specific functionality of a subsystem/class of devices.



## Name

struct class — device classes

## Synopsis

```
struct class {
    const char * name;
    struct module * owner;
    struct class_attribute * class_attrs;
    const struct attribute_group ** dev_groups;
    struct kobject * dev_kobj;
    int (* dev_uevent) (struct device *dev, struct kobj_uevent_env *env);
    char *(* devnode) (struct device *dev, umode_t *mode);
    void (* class_release) (struct class *class);
    void (* dev_release) (struct device *dev);
    int (* suspend) (struct device *dev, pm_message_t state);
    int (* resume) (struct device *dev);
    const struct kobj_ns_type_operations * ns_type;
    const void *(* namespace) (struct device *dev);
    const struct dev_pm_ops * pm;
    struct subsys_private * p;
};
```

## Members

name	Name of the class.
owner	The module owner.
class_attrs	Default attributes of this class.
dev_groups	Default attributes of the devices that belong to the class.
dev_kobj	The kobject that represents this class and links it into the hierarchy.
dev_uevent	Called when a device is added, removed from this class, or a few other things that generate uevents to add the environment variables.
devnode	Callback to provide the devtmpfs.
class_release	Called to release this class.
dev_release	Called to release the device.
suspend	Used to put the device to sleep mode, usually to a low power state.
resume	Used to bring the device from the sleep mode.
ns_type	Callbacks so sysfs can determine namespaces.
namespace	Namespace of the device belongs to this class.
pm	The default device power management operations of this class.
p	The private data of the driver core, no one other than the driver core can touch this.

## Description

A class is a higher-level view of a device that abstracts out low-level implementation details. Drivers may see a SCSI disk or an ATA disk, but, at the class level, they are all simply disks. Classes allow user space to work with devices based on what they do, rather than how they are connected or how they work.

## Name

struct device — The basic device structure

## Synopsis

```
struct device {
    struct device * parent;
    struct device_private * p;
    struct kobject kobj;
    const char * init_name;
    const struct device_type * type;
    struct mutex mutex;
    struct bus_type * bus;
    struct device_driver * driver;
    void * platform_data;
    void * driver_data;
    struct dev_pm_info power;
    struct dev_pm_domain * pm_domain;
#ifdef CONFIG_GENERIC_MSI_IRQ_DOMAIN
    struct irq_domain * msi_domain;
#endif
#ifdef CONFIG_PINCTRL
    struct dev_pin_info * pins;
#endif
#ifdef CONFIG_GENERIC_MSI_IRQ
    struct list_head msi_list;
#endif
#ifdef CONFIG_NUMA
    int numa_node;
#endif
    u64 * dma_mask;
    u64 coherent_dma_mask;
    unsigned long dma_pfn_offset;
    struct device_dma_parameters * dma_parms;
    struct list_head dma_pools;
    struct dma_coherent_mem * dma_mem;
#ifdef CONFIG_DMA_CMA
    struct cma * cma_area;
#endif
    struct dev_archdata archdata;
    struct device_node * of_node;
    struct fwnode_handle * fwnode;
    dev_t devt;
    u32 id;
    spinlock_t devres_lock;
    struct list_head devres_head;
    struct klist_node knode_class;
    struct class * class;
    const struct attribute_group ** groups;
    void (* release) (struct device *dev);
    struct iommu_group * iommu_group;
    bool offline_disabled:1;
}
```

```
    bool offline:1;
};
```

## Members

parent	The device's “parent” device, the device to which it is attached. In most cases, a parent device is some sort of bus or host controller. If parent is NULL, the device, is a top-level device, which is not usually what you want.
p	Holds the private data of the driver core portions of the device. See the comment of the struct device_private for detail.
kobj	A top-level, abstract class from which other classes are derived.
init_name	Initial name of the device.
type	The type of device. This identifies the device type and carries type-specific information.
mutex	Mutex to synchronize calls to its driver.
bus	Type of bus device is on.
driver	Which driver has allocated this
platform_data	Platform data specific to the device.
driver_data	Private pointer for driver specific info.
power	For device power management. See Documentation/power/devices.txt for details.
pm_domain	Provide callbacks that are executed during system suspend, hibernation, system resume and during runtime PM transitions along with subsystem-level and driver-level callbacks.
msi_domain	The generic MSI domain this device is using.
pins	For device pin management. See Documentation/pinctrl.txt for details.
msi_list	Hosts MSI descriptors
numa_node	NUMA node this device is close to.
dma_mask	Dma mask (if dma'ble device).
coherent_dma_mask	Like dma_mask, but for alloc_coherent mapping as not all hardware supports 64-bit addresses for consistent allocations such descriptors.
dma_pfn_offset	offset of DMA memory range relatively of RAM
dma_parms	A low level driver may set these to teach IOMMU code about segment limitations.
dma_pools	Dma pools (if dma'ble device).
dma_mem	Internal for coherent mem override.

<code>cma_area</code>	Contiguous memory area for dma allocations
<code>archdata</code>	For arch-specific additions.
<code>of_node</code>	Associated device tree node.
<code>fwnode</code>	Associated device node supplied by platform firmware.
<code>devt</code>	For creating the sysfs “dev”.
<code>id</code>	device instance
<code>devres_lock</code>	Spinlock to protect the resource of the device.
<code>devres_head</code>	The resources list of the device.
<code>knode_class</code>	The node used to add the device to the class list.
<code>class</code>	The class of the device.
<code>groups</code>	Optional attribute groups.
<code>release</code>	Callback to free the device after all references have gone away. This should be set by the allocator of the device (i.e. the bus driver that discovered the device).
<code>iommu_group</code>	IOMMU group the device belongs to.
<code>offline_disabled</code>	If set, the device is permanently online.
<code>offline</code>	Set after successful invocation of bus type's <code>.offline</code> .

## Example

```
For devices on custom boards, as typical of embedded
and SOC based hardware, Linux often uses platform_data to point
to board-specific structures describing devices and how they
are wired. That can include what ports are available, chip
variants, which GPIO pins act in what additional roles, and so
on. This shrinks the “Board Support Packages” (BSPs) and
minimizes board-specific #ifdefs in drivers.
```

## Description

At the lowest level, every device in a Linux system is represented by an instance of `struct device`. The device structure contains the information that the device model core needs to model the system. Most subsystems, however, track additional information about the devices they host. As a result, it is rare for devices to be represented by bare device structures; instead, that structure, like `kobject` structures, is usually embedded within a higher-level representation of the device.

## Name

`module_driver` — Helper macro for drivers that don't do anything special in module init/exit. This eliminates a lot of boilerplate. Each module may only use this macro once, and calling it replaces `module_init` and `module_exit`.

## Synopsis

```
module_driver ( __driver, __register, __unregister, ... );
```

## Arguments

<i>__driver</i>	driver name
<i>__register</i>	register function for this driver type
<i>__unregister</i>	unregister function for this driver type @...: Additional arguments to be passed to <i>__register</i> and <i>__unregister</i> .
...	variable arguments

## Description

Use this macro to construct bus specific macros for registering drivers, and do not use it on its own.

## Name

`builtin_driver` — Helper macro for drivers that don't do anything special in init and have no exit. This eliminates some boilerplate. Each driver may only use this macro once, and calling it replaces `device_initcall` (or in some cases, the legacy `__initcall`). This is meant to be a direct parallel of `module_driver` above but without the `__exit` stuff that is not used for builtin cases.

## Synopsis

```
builtin_driver ( __driver, __register, ... );
```

## Arguments

*\_\_driver*      driver name

*\_\_register*    register function for this driver type @...: Additional arguments to be passed to `__register`

...            variable arguments

## Description

Use this macro to construct bus specific macros for registering drivers, and do not use it on its own.

# Device Drivers Base

## Name

`driver_init` — initialize driver model.

## Synopsis

```
void driver_init ( void );
```

## Arguments

*void* no arguments

## Description

Call the driver model init functions to initialize their subsystems. Called early from `init/main.c`.



## Name

`driver_for_each_device` — Iterator for devices bound to a driver.

## Synopsis

```
int driver_for_each_device (struct device_driver * drv, struct device  
* start, void * data, int (*fn) (struct device *, void *));
```

## Arguments

*drv*     Driver we're iterating.

*start*   Device to begin with

*data*    Data to pass to the callback.

*fn*      Function to call for each device.

## Description

Iterate over the *drv*'s list of devices calling *fn* for each one.

## Name

`driver_find_device` — device iterator for locating a particular device.

## Synopsis

```
struct device * driver_find_device (struct device_driver * drv, struct
device * start, void * data, int (*match) (struct device *dev, void
*data));
```

## Arguments

*drv*      The device's driver

*start*    Device to begin with

*data*     Data to pass to match function

*match*    Callback function to check device

## Description

This is similar to the `driver_for_each_device` function above, but it returns a reference to a device that is 'found' for later use, as determined by the *match* callback.

The callback should return 0 if the device doesn't match and non-zero if it does. If the callback returns non-zero, this function will return to the caller and not iterate over any more devices.

## Name

`driver_create_file` — create sysfs file for driver.

## Synopsis

```
int driver_create_file (struct device_driver * drv, const struct
driver_attribute * attr);
```

## Arguments

*drv* driver.

*attr* driver attribute descriptor.

## Name

`driver_remove_file` — remove sysfs file for driver.

## Synopsis

```
void driver_remove_file (struct device_driver * drv, const struct  
driver_attribute * attr);
```

## Arguments

*drv* driver.

*attr* driver attribute descriptor.

## Name

`driver_register` — register driver with bus

## Synopsis

```
int driver_register (struct device_driver * drv);
```

## Arguments

*drv*    driver to register

## Description

We pass off most of the work to the `bus_add_driver` call, since most of the things we have to do deal with the bus structures.

## Name

`driver_unregister` — remove driver from system.

## Synopsis

```
void driver_unregister (struct device_driver * drv);
```

## Arguments

*drv* driver.

## Description

Again, we pass off most of the work to the bus-level call.

## Name

`driver_find` — locate driver on a bus by its name.

## Synopsis

```
struct device_driver * driver_find (const char * name, struct bus_type  
* bus);
```

## Arguments

*name*    name of the driver.

*bus*     bus to scan for the driver.

## Description

Call `kset_find_obj` to iterate over list of drivers on a bus to find driver by name. Return driver if found.

This routine provides no locking to prevent the driver it returns from being unregistered or unloaded while the caller is using it. The caller is responsible for preventing this.

## Name

`dev_driver_string` — Return a device's driver name, if at all possible

## Synopsis

```
const char * dev_driver_string (const struct device * dev);
```

## Arguments

*dev*    struct device to get the name of

## Description

Will return the device's driver's name if it is bound to a device. If the device is not bound to a driver, it will return the name of the bus it is attached to. If it is not attached to a bus either, an empty string will be returned.



## Name

`device_create_file` — create sysfs attribute file for device.

## Synopsis

```
int  device_create_file (struct device * dev, const struct
device_attribute * attr);
```

## Arguments

*dev* device.

*attr* device attribute descriptor.

## Name

`device_remove_file` — remove sysfs attribute file.

## Synopsis

```
void device_remove_file (struct device * dev, const struct  
device_attribute * attr);
```

## Arguments

*dev* device.

*attr* device attribute descriptor.

## Name

`device_remove_file_self` — remove sysfs attribute file from its own method.

## Synopsis

```
bool device_remove_file_self (struct device * dev, const struct
device_attribute * attr);
```

## Arguments

*dev* device.

*attr* device attribute descriptor.

## Description

See `kernfs_remove_self` for details.

## Name

`device_create_bin_file` — create sysfs binary attribute file for device.

## Synopsis

```
int  device_create_bin_file (struct device * dev, const struct
bin_attribute * attr);
```

## Arguments

*dev* device.

*attr* device binary attribute descriptor.

## Name

`device_remove_bin_file` — remove sysfs binary attribute file

## Synopsis

```
void device_remove_bin_file (struct device * dev, const struct  
bin_attribute * attr);
```

## Arguments

*dev* device.

*attr* device binary attribute descriptor.

## Name

`device_initialize` — init device structure.

## Synopsis

```
void device_initialize (struct device * dev);
```

## Arguments

*dev* device.

## Description

This prepares the device for use by other layers by initializing its fields. It is the first half of `device_register`, if called by that function, though it can also be called separately, so one may use *dev*'s fields. In particular, `get_device`/`put_device` may be used for reference counting of *dev* after calling this function.

All fields in *dev* must be initialized by the caller to 0, except for those explicitly set to some other value. The simplest approach is to use `kzalloc` to allocate the structure containing *dev*.

## NOTE

Use `put_device` to give up your reference instead of freeing *dev* directly once you have called this function.

## Name

`dev_set_name` — set a device name

## Synopsis

```
int dev_set_name (struct device * dev, const char * fmt, ...);
```

## Arguments

*dev*    device

*fmt*    format string for the device's name

*...*    variable arguments

## Name

`device_add` — add device to device hierarchy.

## Synopsis

```
int device_add (struct device * dev);
```

## Arguments

*dev* device.

## Description

This is part 2 of `device_register`, though may be called separately `_iff_device_initialize` has been called separately.

This adds *dev* to the kobject hierarchy via `kobject_add`, adds it to the global and sibling lists for the device, then adds it to the other relevant subsystems of the driver model.

Do not call this routine or `device_register` more than once for any device structure. The driver model core is not designed to work with devices that get unregistered and then spring back to life. (Among other things, it's very hard to guarantee that all references to the previous incarnation of *dev* have been dropped.) Allocate and register a fresh new struct device instead.

## NOTE

Never directly free *dev* after calling this function, even if it returned an error! Always use `put_device` to give up your reference instead.



## Name

`device_register` — register a device with the system.

## Synopsis

```
int device_register (struct device * dev);
```

## Arguments

*dev* pointer to the device structure

## Description

This happens in two clean steps - initialize the device and add it to the system. The two steps can be called separately, but this is the easiest and most common. I.e. you should only call the two helpers separately if have a clearly defined need to use and refcount the device before it is added to the hierarchy.

For more information, see the kerneldoc for `device_initialize` and `device_add`.

## NOTE

Never directly free *dev* after calling this function, even if it returned an error! Always use `put_device` to give up the reference initialized in this function instead.

## Name

`get_device` — increment reference count for device.

## Synopsis

```
struct device * get_device (struct device * dev);
```

## Arguments

*dev* device.

## Description

This simply forwards the call to `kobject_get`, though we do take care to provide for the case that we get a NULL pointer passed in.

## Name

`put_device` — decrement reference count.

## Synopsis

```
void put_device (struct device * dev);
```

## Arguments

*dev* device in question.

## Name

`device_del` — delete device from system.

## Synopsis

```
void device_del (struct device * dev);
```

## Arguments

*dev* device.

## Description

This is the first part of the device unregistration sequence. This removes the device from the lists we control from here, has it removed from the other driver model subsystems it was added to in `device_add`, and removes it from the kobject hierarchy.

## NOTE

this should be called manually `_iff_ device_add` was also called manually.

## Name

`device_unregister` — unregister device from system.

## Synopsis

```
void device_unregister (struct device * dev);
```

## Arguments

*dev* device going away.

## Description

We do this in two parts, like we do `device_register`. First, we remove it from all the subsystems with `device_del`, then we decrement the reference count via `put_device`. If that is the final reference count, the device will be cleaned up via `device_release` above. Otherwise, the structure will stick around until the final reference to the device is dropped.

## Name

`device_for_each_child` — device child iterator.

## Synopsis

```
int device_for_each_child (struct device * parent, void * data, int
(*fn) (struct device *dev, void *data));
```

## Arguments

*parent* parent struct device.

*data* data for the callback.

*fn* function to be called for each device.

## Description

Iterate over *parent*'s child devices, and call *fn* for each, passing it *data*.

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

## Name

`device_for_each_child_reverse` — device child iterator in reversed order.

## Synopsis

```
int device_for_each_child_reverse (struct device * parent, void * data,
int (*fn) (struct device *dev, void *data));
```

## Arguments

*parent*    parent struct device.

*data*      data for the callback.

*fn*        function to be called for each device.

## Description

Iterate over *parent*'s child devices, and call *fn* for each, passing it *data*.

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

## Name

`device_find_child` — device iterator for locating a particular device.

## Synopsis

```
struct device * device_find_child (struct device * parent, void * data,  
int (*match) (struct device *dev, void *data));
```

## Arguments

*parent*    parent struct device

*data*      Data to pass to match function

*match*     Callback function to check device

## Description

This is similar to the `device_for_each_child` function above, but it returns a reference to a device that is 'found' for later use, as determined by the *match* callback.

The callback should return 0 if the device doesn't match and non-zero if it does. If the callback returns non-zero and a reference to the current device can be obtained, this function will return to the caller and not iterate over any more devices.

## NOTE

you will need to drop the reference with `put_device` after use.



## Name

`__root_device_register` — allocate and register a root device

## Synopsis

```
struct device * __root_device_register (const char * name, struct module  
* owner);
```

## Arguments

*name*     root device name

*owner*    owner module of the root device, usually `THIS_MODULE`

## Description

This function allocates a root device and registers it using `device_register`. In order to free the returned device, use `root_device_unregister`.

Root devices are dummy devices which allow other devices to be grouped under `/sys/devices`. Use this function to allocate a root device and then use it as the parent of any device which should appear under `/sys/devices/{name}`

The `/sys/devices/{name}` directory will also contain a 'module' symlink which points to the *owner* directory in `sysfs`.

Returns struct device pointer on success, or `ERR_PTR` on error.

## Note

You probably want to use `root_device_register`.

## Name

`root_device_unregister` — unregister and free a root device

## Synopsis

```
void root_device_unregister (struct device * dev);
```

## Arguments

*dev*    device going away

## Description

This function unregisters and cleans up a device that was created by `root_device_register`.

## Name

`device_create_vargs` — creates a device and registers it with sysfs

## Synopsis

```
struct device * device_create_vargs (struct class * class, struct device  
* parent, dev_t devt, void * drvdata, const char * fmt, va_list args);
```

## Arguments

<i>class</i>	pointer to the struct class that this device should be registered to
<i>parent</i>	pointer to the parent struct device of this new device, if any
<i>devt</i>	the dev_t for the char device to be added
<i>drvdata</i>	the data to be added to the device for callbacks
<i>fmt</i>	string for the device's name
<i>args</i>	va_list for the device's name

## Description

This function can be used by char device classes. A struct device will be created in sysfs, registered to the specified class.

A “dev” file will be created, showing the dev\_t for the device, if the dev\_t is not 0,0. If a pointer to a parent struct device is passed in, the newly created struct device will be a child of that device in sysfs. The pointer to the struct device will be returned from the call. Any further sysfs files that might be required can be created using this pointer.

Returns struct device pointer on success, or ERR\_PTR on error.

## Note

the struct class passed to this function must have previously been created with a call to `class_create`.

## Name

`device_create` — creates a device and registers it with sysfs

## Synopsis

```
struct device * device_create (struct class * class, struct device *  
parent, dev_t devt, void * drvdata, const char * fmt, ...);
```

## Arguments

<i>class</i>	pointer to the struct class that this device should be registered to
<i>parent</i>	pointer to the parent struct device of this new device, if any
<i>devt</i>	the dev_t for the char device to be added
<i>drvdata</i>	the data to be added to the device for callbacks
<i>fmt</i>	string for the device's name
...	variable arguments

## Description

This function can be used by char device classes. A struct device will be created in sysfs, registered to the specified class.

A “dev” file will be created, showing the dev\_t for the device, if the dev\_t is not 0,0. If a pointer to a parent struct device is passed in, the newly created struct device will be a child of that device in sysfs. The pointer to the struct device will be returned from the call. Any further sysfs files that might be required can be created using this pointer.

Returns struct device pointer on success, or ERR\_PTR on error.

## Note

the struct class passed to this function must have previously been created with a call to `class_create`.

## Name

`device_create_with_groups` — creates a device and registers it with sysfs

## Synopsis

```
struct device * device_create_with_groups (struct class * class, struct
device * parent, dev_t devt, void * drvdata, const struct attribute_group
** groups, const char * fmt, ...);
```

## Arguments

<i>class</i>	pointer to the struct class that this device should be registered to
<i>parent</i>	pointer to the parent struct device of this new device, if any
<i>devt</i>	the dev_t for the char device to be added
<i>drvdata</i>	the data to be added to the device for callbacks
<i>groups</i>	NULL-terminated list of attribute groups to be created
<i>fmt</i>	string for the device's name
...	variable arguments

## Description

This function can be used by char device classes. A struct device will be created in sysfs, registered to the specified class. Additional attributes specified in the `groups` parameter will also be created automatically.

A “dev” file will be created, showing the `dev_t` for the device, if the `dev_t` is not 0,0. If a pointer to a parent struct device is passed in, the newly created struct device will be a child of that device in sysfs. The pointer to the struct device will be returned from the call. Any further sysfs files that might be required can be created using this pointer.

Returns struct device pointer on success, or `ERR_PTR` on error.

## Note

the struct class passed to this function must have previously been created with a call to `class_create`.

## Name

`device_destroy` — removes a device that was created with `device_create`

## Synopsis

```
void device_destroy (struct class * class, dev_t devt);
```

## Arguments

*class*    pointer to the struct class that this device was registered with

*devt*    the dev\_t of the device that was previously registered

## Description

This call unregisters and cleans up a device that was created with a call to `device_create`.

## Name

`device_rename` — renames a device

## Synopsis

```
int device_rename (struct device * dev, const char * new_name);
```

## Arguments

*dev*            the pointer to the struct device to be renamed

*new\_name*    the new name of the device

## Description

It is the responsibility of the caller to provide mutual exclusion between two different calls of `device_rename` on the same device to ensure that `new_name` is valid and won't conflict with other devices.

## Note

Don't call this function. Currently, the networking layer calls this function, but that will change. The following text from Kay Sievers offers

## some insight

Renaming devices is racy at many levels, symlinks and other stuff are not replaced atomically, and you get a “move” uevent, but it's not easy to connect the event to the old and new device. Device nodes are not renamed at all, there isn't even support for that in the kernel now.

In the meantime, during renaming, your target name might be taken by another driver, creating conflicts. Or the old name is taken directly after you renamed it -- then you get events for the same DEVPATH, before you even see the “move” event. It's just a mess, and nothing new should ever rely on kernel device renaming. Besides that, it's not even implemented now for other things than (driver-core wise very simple) network devices.

We are currently about to change network renaming in udev to completely disallow renaming of devices in the same namespace as the kernel uses, because we can't solve the problems properly, that arise with swapping names of multiple interfaces without races. Means, renaming of `eth[0-9]*` will only be allowed to some other name than `eth[0-9]*`, for the aforementioned reasons.

Make up a “real” name in the driver before you register anything, or add some other attributes for userspace to find the device, or use udev to add symlinks -- but never rename kernel devices later, it's a complete mess. We don't even want to get into that and try to implement the missing pieces in the core. We really have other pieces to fix in the driver core mess. :)

## Name

`device_move` — moves a device to a new parent

## Synopsis

```
int device_move (struct device * dev, struct device * new_parent, enum  
dpm_order dpm_order);
```

## Arguments

<i>dev</i>	the pointer to the struct device to be moved
<i>new_parent</i>	the new parent of the device (can be NULL)
<i>dpm_order</i>	how to reorder the <code>dpm_list</code>



## Name

`set_primary_fwnode` — Change the primary firmware node of a given device.

## Synopsis

```
void set_primary_fwnode (struct device * dev, struct fwnode_handle *  
fwnode);
```

## Arguments

*dev*      Device to handle.

*fwnode*   New primary firmware node of the device.

## Description

Set the device's firmware node pointer to *fwnode*, but if a secondary firmware node of the device is present, preserve it.

## Name

`register_syscore_ops` — Register a set of system core operations.

## Synopsis

```
void register_syscore_ops (struct syscore_ops * ops);
```

## Arguments

*ops*   System core operations to register.

## Name

`unregister_syscore_ops` — Unregister a set of system core operations.

## Synopsis

```
void unregister_syscore_ops (struct syscore_ops * ops);
```

## Arguments

*ops*   System core operations to unregister.

## Name

`syscore_suspend` — Execute all the registered system core suspend callbacks.

## Synopsis

```
int syscore_suspend ( void );
```

## Arguments

*void* no arguments

## Description

This function is executed with one CPU on-line and disabled interrupts.

## Name

`syscore_resume` — Execute all the registered system core resume callbacks.

## Synopsis

```
void syscore_resume ( void );
```

## Arguments

*void* no arguments

## Description

This function is executed with one CPU on-line and disabled interrupts.

## Name

`__class_create` — create a struct class structure

## Synopsis

```
struct class * __class_create (struct module * owner, const char * name,  
struct lock_class_key * key);
```

## Arguments

*owner*    pointer to the module that is to “own” this struct class

*name*    pointer to a string for the name of this class.

*key*    the lock\_class\_key for this class; used by mutex lock debugging

## Description

This is used to create a struct class pointer that can then be used in calls to `device_create`.

Returns struct class pointer on success, or `ERR_PTR` on error.

Note, the pointer created here is to be destroyed when finished by making a call to `class_destroy`.

## Name

`class_destroy` — destroys a struct class structure

## Synopsis

```
void class_destroy (struct class * cls);
```

## Arguments

*cls* pointer to the struct class that is to be destroyed

## Description

Note, the pointer to be destroyed must have been created with a call to `class_create`.

## Name

`class_dev_iter_init` — initialize class device iterator

## Synopsis

```
void class_dev_iter_init (struct class_dev_iter * iter, struct class *  
class, struct device * start, const struct device_type * type);
```

## Arguments

*iter*    class iterator to initialize

*class*   the class we wanna iterate over

*start*   the device to start iterating from, if any

*type*    device\_type of the devices to iterate over, NULL for all

## Description

Initialize class iterator *iter* such that it iterates over devices of *class*. If *start* is set, the list iteration will start there, otherwise if it is NULL, the iteration starts at the beginning of the list.



## Name

`class_dev_iter_next` — iterate to the next device

## Synopsis

```
struct device * class_dev_iter_next (struct class_dev_iter * iter);
```

## Arguments

*iter* class iterator to proceed

## Description

Proceed *iter* to the next device and return it. Returns NULL if iteration is complete.

The returned device is referenced and won't be released till iterator is proceed to the next device or exited. The caller is free to do whatever it wants to do with the device including calling back into class code.

## Name

`class_dev_iter_exit` — finish iteration

## Synopsis

```
void class_dev_iter_exit (struct class_dev_iter * iter);
```

## Arguments

*iter* class iterator to finish

## Description

Finish an iteration. Always call this function after iteration is complete whether the iteration ran till the end or not.

## Name

`class_for_each_device` — device iterator

## Synopsis

```
int class_for_each_device (struct class * class, struct device * start,  
void * data, int (*fn) (struct device *, void *));
```

## Arguments

*class*    the class we're iterating

*start*    the device to start with in the list, if any.

*data*    data for the callback

*fn*       function to be called for each device

## Description

Iterate over *class*'s list of devices, and call *fn* for each, passing it *data*. If *start* is set, the list iteration will start there, otherwise if it is NULL, the iteration starts at the beginning of the list.

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

*fn* is allowed to do anything including calling back into class code. There's no locking restriction.

## Name

`class_find_device` — device iterator for locating a particular device

## Synopsis

```
struct device * class_find_device (struct class * class, struct device *  
start, const void * data, int (*match) (struct device *, const void *));
```

## Arguments

*class*    the class we're iterating

*start*    Device to begin with

*data*    data for the match function

*match*   function to check device

## Description

This is similar to the `class_for_each_dev` function above, but it returns a reference to a device that is 'found' for later use, as determined by the *match* callback.

The callback should return 0 if the device doesn't match and non-zero if it does. If the callback returns non-zero, this function will return to the caller and not iterate over any more devices.

Note, you will need to drop the reference with `put_device` after use.

*match* is allowed to do anything including calling back into class code. There's no locking restriction.

## Name

`class_compat_register` — register a compatibility class

## Synopsis

```
struct class_compat * class_compat_register (const char * name);
```

## Arguments

*name* the name of the class

## Description

Compatibility class are meant as a temporary user-space compatibility workaround when converting a family of class devices to a bus devices.

## Name

`class_compat_unregister` — unregister a compatibility class

## Synopsis

```
void class_compat_unregister (struct class_compat * cls);
```

## Arguments

*cls* the class to unregister

## Name

`class_compat_create_link` — create a compatibility class device link to a bus device

## Synopsis

```
int class_compat_create_link (struct class_compat * cls, struct device  
* dev, struct device * device_link);
```

## Arguments

*cls*                    the compatibility class

*dev*                    the target bus device

*device\_link*    an optional device to which a “device” link should be created

## Name

`class_compat_remove_link` — remove a compatibility class device link to a bus device

## Synopsis

```
void class_compat_remove_link (struct class_compat * cls, struct device  
* dev, struct device * device_link);
```

## Arguments

*cls*                    the compatibility class

*dev*                    the target bus device

*device\_link*    an optional device to which a “device” link was previously created



## Name

`unregister_node` — unregister a node device

## Synopsis

```
void unregister_node (struct node * node);
```

## Arguments

*node* node going away

## Description

Unregisters a node device *node*. All the devices on the node must be unregistered before calling this function.

## Name

`request_firmware` — send firmware request and wait for it

## Synopsis

```
int request_firmware (const struct firmware ** firmware_p, const char
* name, struct device * device);
```

## Arguments

<i>firmware_p</i>	pointer to firmware image
<i>name</i>	name of firmware file
<i>device</i>	device for which firmware is being loaded

## Description

*firmware\_p* will be used to return a firmware image by the name of *name* for device *device*.

Should be called from user context where sleeping is allowed.

*name* will be used as `$FIRMWARE` in the uevent environment and should be distinctive enough not to be confused with any other firmware image for this or any other device.

Caller must hold the reference count of *device*.

The function can be called safely inside device's suspend and resume callback.

## Name

`request_firmware_direct` — load firmware directly without usermode helper

## Synopsis

```
int request_firmware_direct (const struct firmware ** firmware_p, const
char * name, struct device * device);
```

## Arguments

<i>firmware_p</i>	pointer to firmware image
<i>name</i>	name of firmware file
<i>device</i>	device for which firmware is being loaded

## Description

This function works pretty much like `request_firmware`, but this doesn't fall back to usermode helper even if the firmware couldn't be loaded directly from fs. Hence it's useful for loading optional firmwares, which aren't always present, without extra long timeouts of udev.

## Name

`release_firmware` — release the resource associated with a firmware image

## Synopsis

```
void release_firmware (const struct firmware * fw);
```

## Arguments

*fw*    firmware resource to release

## Name

`request_firmware_nowait` — asynchronous version of `request_firmware`

## Synopsis

```
int request_firmware_nowait (struct module * module, bool uevent, const
char * name, struct device * device, gfp_t gfp, void * context, void
(*cont) (const struct firmware *fw, void *context));
```

## Arguments

<i>module</i>	module requesting the firmware
<i>uevent</i>	sends uevent to copy the firmware image if this flag is non-zero else the firmware copy must be done manually.
<i>name</i>	name of firmware file
<i>device</i>	device for which firmware is being loaded
<i>gfp</i>	allocation flags
<i>context</i>	will be passed over to <i>cont</i> , and <i>fw</i> may be NULL if firmware request fails.
<i>cont</i>	function will be called asynchronously when the firmware request is over.

## Description

Caller must hold the reference count of *device*.

Asynchronous variant of `request_firmware` for user contexts: - sleep for as small periods as possible since it may increase kernel boot time of built-in device drivers requesting firmware in their `->probe` methods, if *gfp* is `GFP_KERNEL`.

- can't sleep at all if *gfp* is `GFP_ATOMIC`.

## Name

`transport_class_register` — register an initial transport class

## Synopsis

```
int transport_class_register (struct transport_class * tclass);
```

## Arguments

*tclass* a pointer to the transport class structure to be initialised

## Description

The transport class contains an embedded class which is used to identify it. The caller should initialise this structure with zeros and then generic class must have been initialised with the actual transport class unique name. There's a macro `DECLARE_TRANSPORT_CLASS` to do this (declared classes still must be registered).

Returns 0 on success or error on failure.

## Name

`transport_class_unregister` — unregister a previously registered class

## Synopsis

```
void transport_class_unregister (struct transport_class * tclass);
```

## Arguments

*tclass* The transport class to unregister

## Description

Must be called prior to deallocating the memory for the transport class.

## Name

`anon_transport_class_register` — register an anonymous class

## Synopsis

```
int anon_transport_class_register (struct anon_transport_class * atc);
```

## Arguments

*atc* The anon transport class to register

## Description

The anonymous transport class contains both a transport class and a container. The idea of an anonymous class is that it never actually has any device attributes associated with it (and thus saves on container storage). So it can only be used for triggering events. Use `prezero` and then use `DECLARE_ANON_TRANSPORT_CLASS` to initialise the anon transport class storage.



## Name

`anon_transport_class_unregister` — unregister an anon class

## Synopsis

```
void anon_transport_class_unregister (struct anon_transport_class *  
    atc);
```

## Arguments

*atc* Pointer to the anon transport class to unregister

## Description

Must be called prior to deallocating the memory for the anon transport class.

## Name

`transport_setup_device` — declare a new dev for transport class association but don't make it visible yet.

## Synopsis

```
void transport_setup_device (struct device * dev);
```

## Arguments

*dev* the generic device representing the entity being added

## Description

Usually, *dev* represents some component in the HBA system (either the HBA itself or a device remote across the HBA bus). This routine is simply a trigger point to see if any set of transport classes wishes to associate with the added device. This allocates storage for the class device and initialises it, but does not yet add it to the system or add attributes to it (you do this with `transport_add_device`). If you have no need for a separate setup and add operations, use `transport_register_device` (see `transport_class.h`).

## Name

`transport_add_device` — declare a new dev for transport class association

## Synopsis

```
void transport_add_device (struct device * dev);
```

## Arguments

*dev* the generic device representing the entity being added

## Description

Usually, *dev* represents some component in the HBA system (either the HBA itself or a device remote across the HBA bus). This routine is simply a trigger point used to add the device to the system and register attributes for it.

## Name

`transport_configure_device` — configure an already set up device

## Synopsis

```
void transport_configure_device (struct device * dev);
```

## Arguments

*dev* generic device representing device to be configured

## Description

The idea of `configure` is simply to provide a point within the setup process to allow the transport class to extract information from a device after it has been setup. This is used in SCSI because we have to have a setup device to begin using the HBA, but after we send the initial inquiry, we use `configure` to extract the device parameters. The device need not have been added to be configured.

## Name

`transport_remove_device` — remove the visibility of a device

## Synopsis

```
void transport_remove_device (struct device * dev);
```

## Arguments

*dev*    generic device to remove

## Description

This call removes the visibility of the device (to the user from sysfs), but does not destroy it. To eliminate a device entirely you must also call `transport_destroy_device`. If you don't need to do remove and destroy as separate operations, use `transport_unregister_device` (see `transport_class.h`) which will perform both calls for you.

## Name

`transport_destroy_device` — destroy a removed device

## Synopsis

```
void transport_destroy_device (struct device * dev);
```

## Arguments

*dev* device to eliminate from the transport class.

## Description

This call triggers the elimination of storage associated with the transport classdev. Note: all it really does is relinquish a reference to the classdev. The memory will not be freed until the last reference goes to zero. Note also that the classdev retains a reference count on dev, so dev too will remain for as long as the transport class device remains around.

## Name

`device_bind_driver` — bind a driver to one device.

## Synopsis

```
int device_bind_driver (struct device * dev);
```

## Arguments

*dev* device.

## Description

Allow manual attachment of a driver to a device. Caller must have already set *dev->driver*.

Note that this does not modify the bus reference count nor take the bus's `rwsem`. Please verify those are accounted for before calling this. (It is ok to call with no other effort from a driver's `probe` method.)

This function must be called with the device lock held.

## Name

`wait_for_device_probe` —

## Synopsis

```
void wait_for_device_probe ( void );
```

## Arguments

*void* no arguments

## Description

Wait for device probing to be completed.



## Name

`device_attach` — try to attach device to a driver.

## Synopsis

```
int device_attach (struct device * dev);
```

## Arguments

*dev* device.

## Description

Walk the list of drivers that the bus has and call `driver_probe_device` for each pair. If a compatible pair is found, break out and return.

Returns 1 if the device was bound to a driver; 0 if no matching driver was found; -ENODEV if the device is not registered.

When called for a USB interface, `dev->parent` lock must be held.

## Name

`driver_attach` — try to bind driver to devices.

## Synopsis

```
int driver_attach (struct device_driver * drv);
```

## Arguments

*drv* driver.

## Description

Walk the list of devices that the bus has on it and try to match the driver with each one. If `driver_probe_device` returns 0 and the `dev->driver` is set, we've found a compatible pair.

## Name

`device_release_driver` — manually detach device from driver.

## Synopsis

```
void device_release_driver (struct device * dev);
```

## Arguments

*dev* device.

## Description

Manually detach device from driver. When called for a USB interface, *dev*->parent lock must be held.

## Name

`platform_device_register_resndata` — add a platform-level device with resources and platform-specific data

## Synopsis

```
struct platform_device * platform_device_register_resndata (struct
device * parent, const char * name, int id, const struct resource * res,
unsigned int num, const void * data, size_t size);
```

## Arguments

<i>parent</i>	parent device for the device we're adding
<i>name</i>	base name of the device we're adding
<i>id</i>	instance id
<i>res</i>	set of resources that needs to be allocated for the device
<i>num</i>	number of resources
<i>data</i>	platform specific data for this platform device
<i>size</i>	size of platform specific data

## Description

Returns struct `platform_device` pointer on success, or `ERR_PTR` on error.

## Name

`platform_device_register_simple` — add a platform-level device and its resources

## Synopsis

```
struct platform_device * platform_device_register_simple (const char *  
name, int id, const struct resource * res, unsigned int num);
```

## Arguments

*name*    base name of the device we're adding

*id*      instance id

*res*     set of resources that needs to be allocated for the device

*num*     number of resources

## Description

This function creates a simple platform device that requires minimal resource and memory management. Canned release function freeing memory allocated for the device allows drivers using such devices to be unloaded without waiting for the last reference to the device to be dropped.

This interface is primarily intended for use with legacy drivers which probe hardware directly. Because such drivers create sysfs device nodes themselves, rather than letting system infrastructure handle such device enumeration tasks, they don't fully conform to the Linux driver model. In particular, when such drivers are built as modules, they can't be “hotplugged”.

Returns struct `platform_device` pointer on success, or `ERR_PTR` on error.

## Name

`platform_device_register_data` — add a platform-level device with platform-specific data

## Synopsis

```
struct platform_device * platform_device_register_data (struct device
* parent, const char * name, int id, const void * data, size_t size);
```

## Arguments

<i>parent</i>	parent device for the device we're adding
<i>name</i>	base name of the device we're adding
<i>id</i>	instance id
<i>data</i>	platform specific data for this platform device
<i>size</i>	size of platform specific data

## Description

This function creates a simple platform device that requires minimal resource and memory management. Canned release function freeing memory allocated for the device allows drivers using such devices to be unloaded without waiting for the last reference to the device to be dropped.

Returns struct `platform_device` pointer on success, or `ERR_PTR` on error.

## Name

`platform_get_resource` — get a resource for a device

## Synopsis

```
struct resource * platform_get_resource (struct platform_device * dev,  
unsigned int type, unsigned int num);
```

## Arguments

*dev*    platform device

*type*   resource type

*num*    resource index

## Name

`platform_get_irq` — get an IRQ for a device

## Synopsis

```
int platform_get_irq (struct platform_device * dev, unsigned int num);
```

## Arguments

*dev*   platform device

*num*   IRQ number index



## Name

platform\_get\_resource\_byname — get a resource for a device by name

## Synopsis

```
struct resource * platform_get_resource_byname (struct platform_device  
* dev, unsigned int type, const char * name);
```

## Arguments

*dev*    platform device

*type*   resource type

*name*   resource name

## Name

`platform_get_irq_byname` — get an IRQ for a device by name

## Synopsis

```
int platform_get_irq_byname (struct platform_device * dev, const char  
* name);
```

## Arguments

*dev*    platform device

*name*   IRQ name

## Name

`platform_add_devices` — add a numbers of platform devices

## Synopsis

```
int platform_add_devices (struct platform_device ** devs, int num);
```

## Arguments

*devs*    array of platform devices to add

*num*     number of platform devices in array

## Name

`platform_device_put` — destroy a platform device

## Synopsis

```
void platform_device_put (struct platform_device * pdev);
```

## Arguments

*pdev*    platform device to free

## Description

Free all memory associated with a platform device. This function must only be externally called in error cases. All other usage is a bug.

## Name

`platform_device_alloc` — create a platform device

## Synopsis

```
struct platform_device * platform_device_alloc (const char * name, int
id);
```

## Arguments

*name*    base name of the device we're adding

*id*      instance id

## Description

Create a platform device object which can have other objects attached to it, and which will have attached objects freed when it is released.

## Name

`platform_device_add_resources` — add resources to a platform device

## Synopsis

```
int platform_device_add_resources (struct platform_device * pdev, const  
struct resource * res, unsigned int num);
```

## Arguments

*pdev*    platform device allocated by `platform_device_alloc` to add resources to

*res*     set of resources that needs to be allocated for the device

*num*    number of resources

## Description

Add a copy of the resources to the platform device. The memory associated with the resources will be freed when the platform device is released.

## Name

`platform_device_add_data` — add platform-specific data to a platform device

## Synopsis

```
int platform_device_add_data (struct platform_device * pdev, const void  
* data, size_t size);
```

## Arguments

*pdev* platform device allocated by `platform_device_alloc` to add resources to

*data* platform specific data for this platform device

*size* size of platform specific data

## Description

Add a copy of platform specific data to the platform device's `platform_data` pointer. The memory associated with the platform data will be freed when the platform device is released.

## Name

`platform_device_add` — add a platform device to device hierarchy

## Synopsis

```
int platform_device_add (struct platform_device * pdev);
```

## Arguments

*pdev* platform device we're adding

## Description

This is part 2 of `platform_device_register`, though may be called separately \_iff\_ `pdev` was allocated by `platform_device_alloc`.



## Name

`platform_device_del` — remove a platform-level device

## Synopsis

```
void platform_device_del (struct platform_device * pdev);
```

## Arguments

*pdev*    platform device we're removing

## Description

Note that this function will also release all memory- and port-based resources owned by the device (*dev->resource*). This function must `_only_` be externally called in error cases. All other usage is a bug.

## Name

`platform_device_register` — add a platform-level device

## Synopsis

```
int platform_device_register (struct platform_device * pdev);
```

## Arguments

*pdev* platform device we're adding

## Name

`platform_device_unregister` — unregister a platform-level device

## Synopsis

```
void platform_device_unregister (struct platform_device * pdev);
```

## Arguments

*pdev*   platform device we're unregistering

## Description

Unregistration is done in 2 steps. First we release all resources and remove it from the subsystem, then we drop reference count by calling `platform_device_put`.

## Name

`platform_device_register_full` — add a platform-level device with resources and platform-specific data

## Synopsis

```
struct platform_device * platform_device_register_full (const struct  
platform_device_info * pdevinfo);
```

## Arguments

*pdevinfo* data used to create device

## Description

Returns struct `platform_device` pointer on success, or `ERR_PTR` on error.

## Name

`__platform_driver_register` — register a driver for platform-level devices

## Synopsis

```
int __platform_driver_register (struct platform_driver * drv, struct
module * owner);
```

## Arguments

*drv*     platform driver structure

*owner*   owning module/driver

## Name

`platform_driver_unregister` — unregister a driver for platform-level devices

## Synopsis

```
void platform_driver_unregister (struct platform_driver * drv);
```

## Arguments

*drv* platform driver structure

## Name

`__platform_driver_probe` — register driver for non-hotpluggable device

## Synopsis

```
int __platform_driver_probe (struct platform_driver * drv, int (*probe)
                             (struct platform_device *), struct module * module);
```

## Arguments

*drv*        platform driver structure

*probe*     the driver probe routine, probably from an `__init` section

*module*    module which will be the owner of the driver

## Description

Use this instead of `platform_driver_register` when you know the device is not hotpluggable and has already been registered, and you want to remove its run-once probe infrastructure from memory after the driver has bound to the device.

One typical use for this would be with drivers for controllers integrated into system-on-chip processors, where the controller devices have been configured as part of board setup.

Note that this is incompatible with deferred probing.

Returns zero if the driver registered and bound to a device, else returns a negative error code and with the driver not registered.

## Name

`__platform_create_bundle` — register driver and create corresponding device

## Synopsis

```
struct platform_device * __platform_create_bundle (struct
platform_driver * driver, int (*probe) (struct platform_device *),
struct resource * res, unsigned int n_res, const void * data, size_t
size, struct module * module);
```

## Arguments

*driver* platform driver structure

*probe* the driver probe routine, probably from an `__init` section

*res* set of resources that needs to be allocated for the device

*n\_res* number of resources

*data* platform specific data for this platform device

*size* size of platform specific data

*module* module which will be the owner of the driver

## Description

Use this in legacy-style modules that probe hardware directly and register a single platform device and corresponding platform driver.

Returns struct platform\_device pointer on success, or ERR\_PTR on error.



## Name

`__platform_register_drivers` — register an array of platform drivers

## Synopsis

```
int __platform_register_drivers (struct platform_driver *const *  
drivers, unsigned int count, struct module * owner);
```

## Arguments

*drivers*    an array of drivers to register

*count*     the number of drivers to register

*owner*     module owning the drivers

## Description

Registers platform drivers specified by an array. On failure to register a driver, all previously registered drivers will be unregistered. Callers of this API should use `platform_unregister_drivers` to unregister drivers in the reverse order.

## Returns

0 on success or a negative error code on failure.

## Name

`platform_unregister_drivers` — unregister an array of platform drivers

## Synopsis

```
void platform_unregister_drivers (struct platform_driver *const *  
drivers, unsigned int count);
```

## Arguments

*drivers*    an array of drivers to unregister

*count*     the number of drivers to unregister

## Description

Unregisters platform drivers specified by an array. This is typically used to complement an earlier call to `platform_register_drivers`. Drivers are unregistered in the reverse order in which they were registered.

## Name

`bus_for_each_dev` — device iterator.

## Synopsis

```
int bus_for_each_dev (struct bus_type * bus, struct device * start, void  
* data, int (*fn) (struct device *, void *));
```

## Arguments

*bus*      bus type.

*start*    device to start iterating from.

*data*     data for the callback.

*fn*       function to be called for each device.

## Description

Iterate over *bus*'s list of devices, and call *fn* for each, passing it *data*. If *start* is not NULL, we use that device to begin iterating from.

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

## NOTE

The device that returns a non-zero value is not retained in any way, nor is its refcount incremented. If the caller needs to retain this data, it should do so, and increment the reference count in the supplied callback.

## Name

`bus_find_device` — device iterator for locating a particular device.

## Synopsis

```
struct device * bus_find_device (struct bus_type * bus, struct device  
* start, void * data, int (*match) (struct device *dev, void *data));
```

## Arguments

*bus*     bus type

*start*   Device to begin with

*data*    Data to pass to match function

*match*   Callback function to check device

## Description

This is similar to the `bus_for_each_dev` function above, but it returns a reference to a device that is 'found' for later use, as determined by the *match* callback.

The callback should return 0 if the device doesn't match and non-zero if it does. If the callback returns non-zero, this function will return to the caller and not iterate over any more devices.

## Name

`bus_find_device_by_name` — device iterator for locating a particular device of a specific name

## Synopsis

```
struct device * bus_find_device_by_name (struct bus_type * bus, struct
device * start, const char * name);
```

## Arguments

*bus*      bus type

*start*    Device to begin with

*name*     name of the device to match

## Description

This is similar to the `bus_find_device` function above, but it handles searching by a name automatically, no need to write another `strcmp` matching function.

## Name

`subsys_find_device_by_id` — find a device with a specific enumeration number

## Synopsis

```
struct device * subsys_find_device_by_id (struct bus_type * subsys,
unsigned int id, struct device * hint);
```

## Arguments

*subsys*    subsystem

*id*        index 'id' in struct device

*hint*      device to check first

## Description

Check the hint's next object and if it is a match return it directly, otherwise, fall back to a full list search. Either way a reference for the returned object is taken.

## Name

`bus_for_each_drv` — driver iterator

## Synopsis

```
int bus_for_each_drv (struct bus_type * bus, struct device_driver *  
start, void * data, int (*fn) (struct device_driver *, void *));
```

## Arguments

*bus*      bus we're dealing with.

*start*    driver to start iterating on.

*data*     data to pass to the callback.

*fn*       function to call for each driver.

## Description

This is nearly identical to the device iterator above. We iterate over each driver that belongs to *bus*, and call *fn* for each. If *fn* returns anything but 0, we break out and return it. If *start* is not NULL, we use it as the head of the list.

## NOTE

we don't return the driver that returns a non-zero value, nor do we leave the reference count incremented for that driver. If the caller needs to know that info, it must set it in the callback. It must also be sure to increment the refcount so it doesn't disappear before returning to the caller.

## Name

`bus_rescan_devices` — rescan devices on the bus for possible drivers

## Synopsis

```
int bus_rescan_devices (struct bus_type * bus);
```

## Arguments

*bus* the bus to scan.

## Description

This function will look for devices on the bus with no driver attached and rescan it against existing drivers to see if it matches any by calling `device_attach` for the unbound devices.



## Name

`device_reprobe` — remove driver for a device and probe for a new driver

## Synopsis

```
int device_reprobe (struct device * dev);
```

## Arguments

*dev* the device to reprobe

## Description

This function detaches the attached driver (if any) for the given device and restarts the driver probing process. It is intended to use if probing criteria changed during a devices lifetime and driver attachment should change accordingly.

## Name

`bus_register` — register a driver-core subsystem

## Synopsis

```
int bus_register (struct bus_type * bus);
```

## Arguments

*bus*    bus to register

## Description

Once we have that, we register the bus with the kobject infrastructure, then register the children subsystems it has: the devices and drivers that belong to the subsystem.

## Name

`bus_unregister` — remove a bus from the system

## Synopsis

```
void bus_unregister (struct bus_type * bus);
```

## Arguments

*bus* bus.

## Description

Unregister the child subsystems and the bus itself. Finally, we call `bus_put` to release the refcount

## Name

`subsys_dev_iter_init` — initialize subsys device iterator

## Synopsis

```
void subsys_dev_iter_init (struct subsys_dev_iter * iter, struct  
bus_type * subsys, struct device * start, const struct device_type *  
type);
```

## Arguments

*iter*      subsys iterator to initialize

*subsys*    the subsys we wanna iterate over

*start*     the device to start iterating from, if any

*type*      device\_type of the devices to iterate over, NULL for all

## Description

Initialize subsys iterator *iter* such that it iterates over devices of *subsys*. If *start* is set, the list iteration will start there, otherwise if it is NULL, the iteration starts at the beginning of the list.

## Name

`subsys_dev_iter_next` — iterate to the next device

## Synopsis

```
struct device * subsys_dev_iter_next (struct subsys_dev_iter * iter);
```

## Arguments

*iter* subsys iterator to proceed

## Description

Proceed *iter* to the next device and return it. Returns NULL if iteration is complete.

The returned device is referenced and won't be released till iterator is proceed to the next device or exited. The caller is free to do whatever it wants to do with the device including calling back into subsys code.

## Name

`subsys_dev_iter_exit` — finish iteration

## Synopsis

```
void subsys_dev_iter_exit (struct subsys_dev_iter * iter);
```

## Arguments

*iter* subsys iterator to finish

## Description

Finish an iteration. Always call this function after iteration is complete whether the iteration ran till the end or not.

## Name

`subsys_system_register` — register a subsystem at `/sys/devices/system/`

## Synopsis

```
int subsys_system_register (struct bus_type * subsys, const struct
attribute_group ** groups);
```

## Arguments

*subsys*    system subsystem

*groups*    default attributes for the root device

## Description

All 'system' subsystems have a `/sys/devices/system/<name>` root device with the name of the subsystem. The root device can carry subsystem- wide attributes. All registered devices are below this single root device and are named after the subsystem with a simple enumeration number appended. The registered devices are not explicitly named; only 'id' in the device needs to be set.

Do not use this interface for anything new, it exists for compatibility with bad ideas only. New subsystems should use plain subsystems; and add the subsystem-wide attributes should be added to the subsystem directory itself and not some create fake root-device placed in `/sys/devices/system/<name>`.

## Name

`subsys_virtual_register` — register a subsystem at `/sys/devices/virtual/`

## Synopsis

```
int subsys_virtual_register (struct bus_type * subsys, const struct
attribute_group ** groups);
```

## Arguments

*subsys*    virtual subsystem

*groups*    default attributes for the root device

## Description

All 'virtual' subsystems have a `/sys/devices/system/<name>` root device with the name of the subsystem. The root device can carry subsystem-wide attributes. All registered devices are below this single root device. There's no restriction on device naming. This is for kernel software constructs which need sysfs interface.

# Device Drivers DMA Management



## Name

`dma_buf_export` — Creates a new `dma_buf`, and associates an anon file with this buffer, so it can be exported. Also connect the allocator specific data and ops to the buffer. Additionally, provide a name string for exporter; useful in debugging.

## Synopsis

```
struct dma_buf * dma_buf_export (const struct dma_buf_export_info *  
exp_info);
```

## Arguments

*exp\_info* [in] holds all the export related information provided by the exporter. see struct `dma_buf_export_info` for further details.

## Description

Returns, on success, a newly created `dma_buf` object, which wraps the supplied private data and operations for `dma_buf_ops`. On either missing ops, or error in allocating struct `dma_buf`, will return negative error.

## Name

`dma_buf_fd` — returns a file descriptor for the given `dma_buf`

## Synopsis

```
int dma_buf_fd (struct dma_buf * dmabuf, int flags);
```

## Arguments

*dmabuf* [in] pointer to `dma_buf` for which fd is required.

*flags* [in] flags to give to fd

## Description

On success, returns an associated 'fd'. Else, returns error.

## Name

`dma_buf_get` — returns the `dma_buf` structure related to an `fd`

## Synopsis

```
struct dma_buf * dma_buf_get (int fd);
```

## Arguments

*fd* [in] `fd` associated with the `dma_buf` to be returned

## Description

On success, returns the `dma_buf` structure associated with an `fd`; uses file's refcounting done by `fget` to increase refcount. returns `ERR_PTR` otherwise.

## Name

`dma_buf_put` — decreases refcount of the buffer

## Synopsis

```
void dma_buf_put (struct dma_buf * dmabuf);
```

## Arguments

*dmabuf* [in] buffer to reduce refcount of

## Description

Uses file's refcounting done implicitly by `fput`

## Name

`dma_buf_attach` — Add the device to `dma_buf`'s attachments list; optionally, calls `attach` of `dma_buf_ops` to allow device-specific attach functionality

## Synopsis

```
struct dma_buf_attachment * dma_buf_attach (struct dma_buf * dmabuf,  
struct device * dev);
```

## Arguments

*dmabuf* [in] buffer to attach device to.

*dev* [in] device to be attached.

## Description

Returns `struct dma_buf_attachment *` for this attachment; returns `ERR_PTR` on error.

## Name

`dma_buf_detach` — Remove the given attachment from `dmabuf`'s attachments list; optionally calls `detach` of `dma_buf_ops` for device-specific detach

## Synopsis

```
void dma_buf_detach (struct dma_buf * dmabuf, struct dma_buf_attachment  
* attach);
```

## Arguments

*dmabuf* [in] buffer to detach from.

*attach* [in] attachment to be detached; is free'd after this call.

## Name

`dma_buf_map_attachment` — Returns the scatterlist table of the attachment; mapped into `_device_` address space. Is a wrapper for `map_dma_buf` of the `dma_buf_ops`.

## Synopsis

```
struct sg_table * dma_buf_map_attachment (struct dma_buf_attachment *  
attach, enum dma_data_direction direction);
```

## Arguments

*attach*        [in] attachment whose scatterlist is to be returned

*direction*   [in] direction of DMA transfer

## Description

Returns `sg_table` containing the scatterlist to be returned; returns `ERR_PTR` on error.

## Name

`dma_buf_unmap_attachment` — unmaps and decreases usecount of the buffer; might deallocate the scatterlist associated. Is a wrapper for `unmap_dma_buf` of `dma_buf_ops`.

## Synopsis

```
void dma_buf_unmap_attachment (struct dma_buf_attachment * attach,  
struct sg_table * sg_table, enum dma_data_direction direction);
```

## Arguments

*attach*        [in] attachment to unmap buffer from

*sg\_table*     [in] scatterlist info of the buffer to unmap

*direction*    [in] direction of DMA transfer



## Name

`dma_buf_begin_cpu_access` — Must be called before accessing a `dma_buf` from the cpu in the kernel context. Calls `begin_cpu_access` to allow exporter-specific preparations. Coherency is only guaranteed in the specified range for the specified access direction.

## Synopsis

```
int dma_buf_begin_cpu_access (struct dma_buf * dmabuf, size_t start,
size_t len, enum dma_data_direction direction);
```

## Arguments

<i>dmabuf</i>	[in] buffer to prepare cpu access for.
<i>start</i>	[in] start of range for cpu access.
<i>len</i>	[in] length of range for cpu access.
<i>direction</i>	[in] length of range for cpu access.

## Description

Can return negative error values, returns 0 on success.

## Name

`dma_buf_end_cpu_access` — Must be called after accessing a `dma_buf` from the cpu in the kernel context. Calls `end_cpu_access` to allow exporter-specific actions. Coherency is only guaranteed in the specified range for the specified access direction.

## Synopsis

```
void dma_buf_end_cpu_access (struct dma_buf * dmabuf, size_t start,  
size_t len, enum dma_data_direction direction);
```

## Arguments

<i>dmabuf</i>	[in] buffer to complete cpu access for.
<i>start</i>	[in] start of range for cpu access.
<i>len</i>	[in] length of range for cpu access.
<i>direction</i>	[in] length of range for cpu access.

## Description

This call must always succeed.

## Name

`dma_buf_kmap_atomic` — Map a page of the buffer object into kernel address space. The same restrictions as for `kmap_atomic` and friends apply.

## Synopsis

```
void * dma_buf_kmap_atomic (struct dma_buf * dmabuf, unsigned long
                             page_num);
```

## Arguments

*dmabuf*      [in] buffer to map page from.

*page\_num*   [in] page in PAGE\_SIZE units to map.

## Description

This call must always succeed, any necessary preparations that might fail need to be done in `begin_cpu_access`.

## Name

`dma_buf_kunmap_atomic` — Unmap a page obtained by `dma_buf_kmap_atomic`.

## Synopsis

```
void dma_buf_kunmap_atomic (struct dma_buf * dmabuf, unsigned long  
    page_num, void * vaddr);
```

## Arguments

*dmabuf*      [in] buffer to unmap page from.

*page\_num*   [in] page in PAGE\_SIZE units to unmap.

*vaddr*      [in] kernel space pointer obtained from `dma_buf_kmap_atomic`.

## Description

This call must always succeed.

## Name

`dma_buf_kmap` — Map a page of the buffer object into kernel address space. The same restrictions as for `kmap` and friends apply.

## Synopsis

```
void * dma_buf_kmap (struct dma_buf * dmabuf, unsigned long page_num);
```

## Arguments

*dmabuf*      [in] buffer to map page from.

*page\_num*   [in] page in PAGE\_SIZE units to map.

## Description

This call must always succeed, any necessary preparations that might fail need to be done in `begin_cpu_access`.

## Name

`dma_buf_kunmap` — Unmap a page obtained by `dma_buf_kmap`.

## Synopsis

```
void dma_buf_kunmap (struct dma_buf * dmabuf, unsigned long page_num,  
void * vaddr);
```

## Arguments

*dmabuf*      [in] buffer to unmap page from.

*page\_num*   [in] page in PAGE\_SIZE units to unmap.

*vaddr*      [in] kernel space pointer obtained from `dma_buf_kmap`.

## Description

This call must always succeed.

## Name

`dma_buf_mmap` — Setup up a userspace mmap with the given vma

## Synopsis

```
int dma_buf_mmap (struct dma_buf * dmabuf, struct vm_area_struct * vma,
unsigned long pgoff);
```

## Arguments

*dmabuf* [in] buffer that should back the vma

*vma* [in] vma for the mmap

*pgoff* [in] offset in pages where this mmap should start within the dma-buf buffer.

## Description

This function adjusts the passed in vma so that it points at the file of the dma\_buf operation. It also adjusts the starting pgoff and does bounds checking on the size of the vma. Then it calls the exporters mmap function to set up the mapping.

Can return negative error values, returns 0 on success.

## Name

`dma_buf_vmap` — Create virtual mapping for the buffer object into kernel address space. Same restrictions as for `vmap` and friends apply.

## Synopsis

```
void * dma_buf_vmap (struct dma_buf * dmabuf);
```

## Arguments

*dmabuf* [in] buffer to vmap

## Description

This call may fail due to lack of virtual mapping address space. These calls are optional in drivers. The intended use for them is for mapping objects linear in kernel space for high use objects. Please attempt to use `kmap/kunmap` before thinking about these interfaces.

Returns `NULL` on error.



## Name

`dma_buf_vunmap` — Unmap a vmap obtained by `dma_buf_vmap`.

## Synopsis

```
void dma_buf_vunmap (struct dma_buf * dmabuf, void * vaddr);
```

## Arguments

*dmabuf*    [in] buffer to vunmap

*vaddr*    [in] vmap to vunmap

## Name

`fence_context_alloc` — allocate an array of fence contexts

## Synopsis

```
unsigned fence_context_alloc (unsigned num);
```

## Arguments

*num* [in] amount of contexts to allocate

## Description

This function will return the first index of the number of fences allocated. The fence context is used for setting fence->context to a unique number.

## Name

`fence_signal_locked` — signal completion of a fence

## Synopsis

```
int fence_signal_locked (struct fence * fence);
```

## Arguments

*fence*    the fence to signal

## Description

Signal completion for software callbacks on a fence, this will unblock `fence_wait` calls and run all the callbacks added with `fence_add_callback`. Can be called multiple times, but since a fence can only go from unsignaled to signaled state, it will only be effective the first time.

Unlike `fence_signal`, this function must be called with `fence->lock` held.

## Name

`fence_signal` — signal completion of a fence

## Synopsis

```
int fence_signal (struct fence * fence);
```

## Arguments

*fence*    the fence to signal

## Description

Signal completion for software callbacks on a fence, this will unblock `fence_wait` calls and run all the callbacks added with `fence_add_callback`. Can be called multiple times, but since a fence can only go from unsignaled to signaled state, it will only be effective the first time.

## Name

`fence_wait_timeout` — sleep until the fence gets signaled or until timeout elapses

## Synopsis

```
signed long fence_wait_timeout (struct fence * fence, bool intr, signed  
long timeout);
```

## Arguments

*fence*      [in] the fence to wait on

*intr*       [in] if true, do an interruptible wait

*timeout*    [in] timeout value in jiffies, or MAX\_SCHEDULE\_TIMEOUT

## Description

Returns `-ERESTARTSYS` if interrupted, 0 if the wait timed out, or the remaining timeout in jiffies on success. Other error values may be returned on custom implementations.

Performs a synchronous wait on this fence. It is assumed the caller directly or indirectly (buf-mgr between reservation and committing) holds a reference to the fence, otherwise the fence might be freed before return, resulting in undefined behavior.

## Name

`fence_enable_sw_signaling` — enable signaling on fence

## Synopsis

```
void fence_enable_sw_signaling (struct fence * fence);
```

## Arguments

*fence* [in] the fence to enable

## Description

this will request for sw signaling to be enabled, to make the fence complete as soon as possible

## Name

`fence_add_callback` — add a callback to be called when the fence is signaled

## Synopsis

```
int fence_add_callback (struct fence * fence, struct fence_cb * cb,
fence_func_t func);
```

## Arguments

*fence* [in] the fence to wait on

*cb* [in] the callback to register

*func* [in] the function to call

## Description

`cb` will be initialized by `fence_add_callback`, no initialization by the caller is required. Any number of callbacks can be registered to a fence, but a callback can only be registered to one fence at a time.

Note that the callback can be called from an atomic context. If fence is already signaled, this function will return `-ENOENT` (and *not* call the callback)

Add a software callback to the fence. Same restrictions apply to refcount as it does to `fence_wait`, however the caller doesn't need to

## keep a refcount to fence afterwards

when software access is enabled, the creator of the fence is required to keep the fence alive until after it signals with `fence_signal`. The callback itself can be called from irq context.

## Name

`fence_remove_callback` — remove a callback from the signaling list

## Synopsis

```
bool fence_remove_callback (struct fence * fence, struct fence_cb * cb);
```

## Arguments

*fence*    [in] the fence to wait on

*cb*        [in] the callback to remove

## Description

Remove a previously queued callback from the fence. This function returns true if the callback is successfully removed, or false if the fence has already been signaled.

**\*WARNING\***: Cancelling a callback should only be done if you really know what you're doing, since deadlocks and race conditions could occur all too easily. For this reason, it should only ever be done on hardware lockup recovery, with a reference held to the fence.



## Name

`fence_default_wait` — default sleep until the fence gets signaled or until timeout elapses

## Synopsis

```
signed long fence_default_wait (struct fence * fence, bool intr, signed  
long timeout);
```

## Arguments

*fence*      [in] the fence to wait on

*intr*        [in] if true, do an interruptible wait

*timeout*    [in] timeout value in jiffies, or MAX\_SCHEDULE\_TIMEOUT

## Description

Returns `-ERESTARTSYS` if interrupted, 0 if the wait timed out, or the remaining timeout in jiffies on success.

## Name

`fence_wait_any_timeout` — sleep until any fence gets signaled or until timeout elapses

## Synopsis

```
signed long fence_wait_any_timeout (struct fence ** fences, uint32_t
count, bool intr, signed long timeout);
```

## Arguments

*fences* [in] array of fences to wait on

*count* [in] number of fences to wait on

*intr* [in] if true, do an interruptible wait

*timeout* [in] timeout value in jiffies, or MAX\_SCHEDULE\_TIMEOUT

## Description

Returns -EINVAL on custom fence wait implementation, -ERESTARTSYS if interrupted, 0 if the wait timed out, or the remaining timeout in jiffies on success.

Synchronous waits for the first fence in the array to be signaled. The caller needs to hold a reference to all fences in the array, otherwise a fence might be freed before return, resulting in undefined behavior.

## Name

`fence_init` — Initialize a custom fence.

## Synopsis

```
void fence_init (struct fence * fence, const struct fence_ops * ops,  
spinlock_t * lock, unsigned context, unsigned seqno);
```

## Arguments

<i>fence</i>	[in] the fence to initialize
<i>ops</i>	[in] the fence_ops for operations on this fence
<i>lock</i>	[in] the irqsafe spinlock to use for locking this fence
<i>context</i>	[in] the execution context this fence is run on
<i>seqno</i>	[in] a linear increasing sequence number for this context

## Description

Initializes an allocated fence, the caller doesn't have to keep its refcount after committing with this fence, but it will need to hold a refcount again if `fence_ops.enable_signaling` gets called. This can be used for other implementing other types of fence.

`context` and `seqno` are used for easy comparison between fences, allowing to check which fence is later by simply using `fence_later`.

## Name

drivers/dma-buf/seqno-fence.c — Document generation inconsistency

## Oops

### Warning

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

## Name

struct fence — software synchronization primitive

## Synopsis

```
struct fence {
    struct kref refcount;
    const struct fence_ops * ops;
    struct rcu_head rcu;
    struct list_head cb_list;
    spinlock_t * lock;
    unsigned context;
    unsigned seqno;
    unsigned long flags;
    ktime_t timestamp;
    int status;
};
```

## Members

refcount	refcount for this fence
ops	fence_ops associated with this fence
rcu	used for releasing fence with kfree_rcu
cb_list	list of all callbacks to call
lock	spin_lock_irqsave used for locking
context	execution context this fence belongs to, returned by fence_context_alloc
seqno	the sequence number of this fence inside the execution context, can be compared to decide which fence would be signaled later.
flags	A mask of FENCE_FLAG_* defined below
timestamp	Timestamp when the fence was signaled.
status	Optional, only valid if < 0, must be set before calling fence_signal, indicates that the fence has completed with an error.

## Description

the flags member must be manipulated and read using the appropriate atomic ops (bit\_\*), so taking the spinlock will not be needed most of the time.

FENCE\_FLAG\_SIGNALED\_BIT - fence is already signaled  
FENCE\_FLAG\_ENABLE\_SIGNAL\_BIT - enable\_signaling might have been called  
FENCE\_FLAG\_USER\_BITS - start of the unused bits, can be used by the implementer of the fence for its own purposes. Can be used in different ways by different fence implementers, so do not rely on this.

\*) Since atomic bitops are used, this is not guaranteed to be the case. Particularly, if the bit was set, but fence\_signal was called right before this bit was set, it would have been able to

set the `FENCE_FLAG_SIGNALED_BIT`, before `enable_signaling` was called. Adding a check for `FENCE_FLAG_SIGNALED_BIT` after setting `FENCE_FLAG_ENABLE_SIGNAL_BIT` closes this race, and makes sure that after `fence_signal` was called, any `enable_signaling` call will have either been completed, or never called at all.

## Name

struct fence\_cb — callback for fence\_add\_callback

## Synopsis

```
struct fence_cb {  
    struct list_head node;  
    fence_func_t func;  
};
```

## Members

node    used by fence\_add\_callback to append this struct to fence::cb\_list

func    fence\_func\_t to call

## Description

This struct will be initialized by fence\_add\_callback, additional data can be passed along by embedding fence\_cb in another struct.

## Name

struct fence\_ops — operations implemented for fence

## Synopsis

```
struct fence_ops {
    const char * (* get_driver_name) (struct fence *fence);
    const char * (* get_timeline_name) (struct fence *fence);
    bool (* enable_signaling) (struct fence *fence);
    bool (* signaled) (struct fence *fence);
    signed long (* wait) (struct fence *fence, bool intr, signed long timeout);
    void (* release) (struct fence *fence);
    int (* fill_driver_data) (struct fence *fence, void *data, int size);
    void (* fence_value_str) (struct fence *fence, char *str, int size);
    void (* timeline_value_str) (struct fence *fence, char *str, int size);
};
```

## Members

get_driver_name	returns the driver name.
get_timeline_name	return the name of the context this fence belongs to.
enable_signaling	enable software signaling of fence.
signaled	[optional] peek whether the fence is signaled, can be null.
wait	custom wait implementation, or fence_default_wait.
release	[optional] called on destruction of fence, can be null
fill_driver_data	[optional] callback to fill in free-form debug info Returns amount of bytes filled, or -errno.
fence_value_str	[optional] fills in the value of the fence as a string
timeline_value_str	[optional] fills in the current value of the timeline as a string

## Notes on enable\_signaling

For fence implementations that have the capability for hw->hw signaling, they can implement this op to enable the necessary irqs, or insert commands into cmdstream, etc. This is called in the first wait or add\_callback path to let the fence implementation know that there is another driver waiting on the signal (ie. hw->sw case).

This function can be called called from atomic context, but not from irq context, so normal spinlocks can be used.

A return value of false indicates the fence already passed, or some failure occurred that made it impossible to enable signaling. True indicates successful enabling.

fence->status may be set in enable\_signaling, but only when false is returned.

Calling fence\_signal before enable\_signaling is called allows for a tiny race window in which enable\_signaling is called during, before, or after fence\_signal. To fight this, it is recommended that before



`enable_signaling` returns true an extra reference is taken on the fence, to be released when the fence is signaled. This will mean `fence_signal` will still be called twice, but the second time will be a noop since it was already signaled.

## Notes on signaled

May set `fence->status` if returning true.

## Notes on wait

Must not be NULL, set to `fence_default_wait` for default implementation. the `fence_default_wait` implementation should work for any fence, as long as `enable_signaling` works correctly.

Must return `-ERESTARTSYS` if the wait is `intr = true` and the wait was interrupted, and remaining jiffies if fence has signaled, or 0 if wait timed out. Can also return other error values on custom implementations, which should be treated as if the fence is signaled. For example a hardware lockup could be reported like that.

## Notes on release

Can be NULL, this function allows additional commands to run on destruction of the fence. Can be called from irq context. If pointer is set to NULL, `kfree` will get called instead.

## Name

`fence_get` — increases refcount of the fence

## Synopsis

```
struct fence * fence_get (struct fence * fence);
```

## Arguments

*fence* [in] fence to increase refcount of

## Description

Returns the same fence, with refcount increased by 1.

## Name

`fence_get_rcu` — get a fence from a `reservation_object_list` with rcu read lock

## Synopsis

```
struct fence * fence_get_rcu (struct fence * fence);
```

## Arguments

*fence* [in] fence to increase refcount of

## Description

Function returns NULL if no refcount could be obtained, or the fence.

## Name

`fence_put` — decreases refcount of the fence

## Synopsis

```
void fence_put (struct fence * fence);
```

## Arguments

*fence* [in] fence to reduce refcount of

## Name

`fence_is_signaled_locked` — Return an indication if the fence is signaled yet.

## Synopsis

```
bool fence_is_signaled_locked (struct fence * fence);
```

## Arguments

*fence* [in] the fence to check

## Description

Returns true if the fence was already signaled, false if not. Since this function doesn't enable signaling, it is not guaranteed to ever return true if `fence_add_callback`, `fence_wait` or `fence_enable_sw_signaling` haven't been called before.

This function requires `fence->lock` to be held.

## Name

`fence_is_signaled` — Return an indication if the fence is signaled yet.

## Synopsis

```
bool fence_is_signaled (struct fence * fence);
```

## Arguments

*fence* [in] the fence to check

## Description

Returns true if the fence was already signaled, false if not. Since this function doesn't enable signaling, it is not guaranteed to ever return true if `fence_add_callback`, `fence_wait` or `fence_enable_sw_signaling` haven't been called before.

It's recommended for seqno fences to call `fence_signal` when the operation is complete, it makes it possible to prevent issues from wraparound between time of issue and time of use by checking the return value of this function before calling hardware-specific wait instructions.

## Name

`fence_is_later` — return if `f1` is chronologically later than `f2`

## Synopsis

```
bool fence_is_later (struct fence * f1, struct fence * f2);
```

## Arguments

*f1* [in] the first fence from the same context

*f2* [in] the second fence from the same context

## Description

Returns true if `f1` is chronologically later than `f2`. Both fences must be from the same context, since a seqno is not re-used across contexts.

## Name

`fence_later` — return the chronologically later fence

## Synopsis

```
struct fence * fence_later (struct fence * f1, struct fence * f2);
```

## Arguments

*f1* [in] the first fence from the same context

*f2* [in] the second fence from the same context

## Description

Returns NULL if both fences are signaled, otherwise the fence that would be signaled last. Both fences must be from the same context, since a seqno is not re-used across contexts.



## Name

`fence_wait` — sleep until the fence gets signaled

## Synopsis

```
signed long fence_wait (struct fence * fence, bool intr);
```

## Arguments

*fence* [in] the fence to wait on

*intr* [in] if true, do an interruptible wait

## Description

This function will return `-ERESTARTSYS` if interrupted by a signal, or 0 if the fence was signaled. Other error values may be returned on custom implementations.

Performs a synchronous wait on this fence. It is assumed the caller directly or indirectly holds a reference to the fence, otherwise the fence might be freed before return, resulting in undefined behavior.

## Name

`to_seqno_fence` — cast a fence to a `seqno_fence`

## Synopsis

```
struct seqno_fence * to_seqno_fence (struct fence * fence);
```

## Arguments

*fence*    fence to cast to a `seqno_fence`

## Description

Returns NULL if the fence is not a `seqno_fence`, or the `seqno_fence` otherwise.

## Name

`seqno_fence_init` — initialize a seqno fence

## Synopsis

```
void seqno_fence_init (struct seqno_fence * fence, spinlock_t * lock,  
struct dma_buf * sync_buf, uint32_t context, uint32_t seqno_ofs,  
uint32_t seqno, enum seqno_fence_condition cond, const struct fence_ops  
* ops);
```

## Arguments

<i>fence</i>	seqno_fence to initialize
<i>lock</i>	pointer to spinlock to use for fence
<i>sync_buf</i>	buffer containing the memory location to signal on
<i>context</i>	the execution context this fence is a part of
<i>seqno_ofs</i>	the offset within <i>sync_buf</i>
<i>seqno</i>	the sequence # to signal on
<i>cond</i>	fence wait condition
<i>ops</i>	the fence_ops for operations on this seqno fence

## Description

This function initializes a struct `seqno_fence` with passed parameters, and takes a reference on `sync_buf` which is released on fence destruction.

A `seqno_fence` is a `dma_fence` which can complete in software when `enable_signaling` is called, but it also completes when `(s32)((sync_buf)[seqno_ofs] - seqno) >= 0` is true

The `seqno_fence` will take a refcount on the `sync_buf` until it's destroyed, but actual lifetime of `sync_buf` may be longer if one of the callers take a reference to it.

Certain hardware have instructions to insert this type of wait condition in the command stream, so no intervention from software would be needed. This type of fence can be destroyed before completed, however a reference on the `sync_buf` dma-buf can be taken. It is encouraged to re-use the same dma-buf for `sync_buf`, since mapping or unmapping the `sync_buf` to the device's vm can be expensive.

It is recommended for creators of `seqno_fence` to call `fence_signal` before destruction. This will prevent possible issues from wraparound at time of issue vs time of check, since users can check `fence_is_signaled` before submitting instructions for the hardware to wait on the fence. However, when `ops.enable_signaling` is not called, it doesn't have to be done as soon as possible, just before there's any real danger of seqno wraparound.

## Name

drivers/dma-buf/reservation.c — Document generation inconsistency

## Oops

### Warning

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

## Name

include/linux/reservation.h — Document generation inconsistency

## Oops

### Warning

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

## Name

`dma_alloc_from_coherent` — try to allocate memory from the per-device coherent area

## Synopsis

```
int dma_alloc_from_coherent (struct device * dev, ssize_t size,
dma_addr_t * dma_handle, void ** ret);
```

## Arguments

<i>dev</i>	device from which we allocate memory
<i>size</i>	size of requested memory area
<i>dma_handle</i>	This will be filled with the correct dma handle
<i>ret</i>	This pointer will be filled with the virtual address to allocated area.

## Description

This function should be only called from per-arch `dma_alloc_coherent` to support allocation from per-device coherent memory pools.

Returns 0 if `dma_alloc_coherent` should continue with allocating from generic memory areas, or !0 if `dma_alloc_coherent` should return *ret*.

## Name

`dma_release_from_coherent` — try to free the memory allocated from per-device coherent memory pool

## Synopsis

```
int dma_release_from_coherent (struct device * dev, int order, void *  
vaddr);
```

## Arguments

*dev*     device from which the memory was allocated

*order*   the order of pages allocated

*vaddr*   virtual address of allocated pages

## Description

This checks whether the memory was allocated from the per-device coherent memory pool and if so, releases that memory.

Returns 1 if we correctly released the memory, or 0 if `dma_release_coherent` should proceed with releasing memory from generic pools.

## Name

`dma_mmap_from_coherent` — try to mmap the memory allocated from per-device coherent memory pool to userspace

## Synopsis

```
int dma_mmap_from_coherent (struct device * dev, struct vm_area_struct  
* vma, void * vaddr, size_t size, int * ret);
```

## Arguments

<i>dev</i>	device from which the memory was allocated
<i>vma</i>	vm_area for the userspace memory
<i>vaddr</i>	cpu address returned by <code>dma_alloc_from_coherent</code>
<i>size</i>	size of the memory buffer allocated by <code>dma_alloc_from_coherent</code>
<i>ret</i>	result from <code>remap_pfn_range</code>

## Description

This checks whether the memory was allocated from the per-device coherent memory pool and if so, maps that memory to the provided `vma`.

Returns 1 if we correctly mapped the memory, or 0 if the caller should proceed with mapping memory from generic pools.



## Name

`dmam_alloc_coherent` — Managed `dma_alloc_coherent`

## Synopsis

```
void * dmam_alloc_coherent (struct device * dev, size_t size, dma_addr_t  
* dma_handle, gfp_t gfp);
```

## Arguments

<i>dev</i>	Device to allocate coherent memory for
<i>size</i>	Size of allocation
<i>dma_handle</i>	Out argument for allocated DMA handle
<i>gfp</i>	Allocation flags

## Description

Managed `dma_alloc_coherent`. Memory allocated using this function will be automatically released on driver detach.

## RETURNS

Pointer to allocated memory on success, NULL on failure.

## Name

`dmam_free_coherent` — Managed `dma_free_coherent`

## Synopsis

```
void dmam_free_coherent (struct device * dev, size_t size, void * vaddr,  
dma_addr_t dma_handle);
```

## Arguments

<i>dev</i>	Device to free coherent memory for
<i>size</i>	Size of allocation
<i>vaddr</i>	Virtual address of the memory to free
<i>dma_handle</i>	DMA handle of the memory to free

## Description

Managed `dma_free_coherent`.

## Name

`dmam_alloc_noncoherent` — Managed `dma_alloc_non_coherent`

## Synopsis

```
void * dmam_alloc_noncoherent (struct device * dev, size_t size,  
dma_addr_t * dma_handle, gfp_t gfp);
```

## Arguments

<i>dev</i>	Device to allocate <code>non_coherent</code> memory for
<i>size</i>	Size of allocation
<i>dma_handle</i>	Out argument for allocated DMA handle
<i>gfp</i>	Allocation flags

## Description

Managed `dma_alloc_non_coherent`. Memory allocated using this function will be automatically released on driver detach.

## RETURNS

Pointer to allocated memory on success, NULL on failure.

## Name

`dmam_free_noncoherent` — Managed `dma_free_noncoherent`

## Synopsis

```
void dmam_free_noncoherent (struct device * dev, size_t size, void *  
vaddr, dma_addr_t dma_handle);
```

## Arguments

<i>dev</i>	Device to free noncoherent memory for
<i>size</i>	Size of allocation
<i>vaddr</i>	Virtual address of the memory to free
<i>dma_handle</i>	DMA handle of the memory to free

## Description

Managed `dma_free_noncoherent`.

## Name

`dmam_declare_coherent_memory` — Managed `dma_declare_coherent_memory`

## Synopsis

```
int dmam_declare_coherent_memory (struct device * dev, phys_addr_t
phys_addr, dma_addr_t device_addr, size_t size, int flags);
```

## Arguments

<i>dev</i>	Device to declare coherent memory for
<i>phys_addr</i>	Physical address of coherent memory to be declared
<i>device_addr</i>	Device address of coherent memory to be declared
<i>size</i>	Size of coherent memory to be declared
<i>flags</i>	Flags

## Description

Managed `dma_declare_coherent_memory`.

## RETURNS

0 on success, -errno on failure.

## Name

`dmam_release_declared_memory` — Managed `dma_release_declared_memory`.

## Synopsis

```
void dmam_release_declared_memory (struct device * dev);
```

## Arguments

*dev* Device to release declared coherent memory for

## Description

Managed `dmam_release_declared_memory`.

# Device Drivers Power Management

## Name

`dpm_resume_start` — Execute “noirq” and “early” device callbacks.

## Synopsis

```
void dpm_resume_start (pm_message_t state);
```

## Arguments

*state*    PM transition of the system being carried out.

## Name

`dpm_resume_end` — Execute “resume” callbacks and complete system transition.

## Synopsis

```
void dpm_resume_end (pm_message_t state);
```

## Arguments

*state* PM transition of the system being carried out.

## Description

Execute “resume” callbacks for all devices and complete the PM transition of the system.



## Name

`dpm_suspend_end` — Execute “late” and “noirq” device suspend callbacks.

## Synopsis

```
int dpm_suspend_end (pm_message_t state);
```

## Arguments

*state*    PM transition of the system being carried out.

## Name

`dpm_suspend_start` — Prepare devices for PM transition and suspend them.

## Synopsis

```
int dpm_suspend_start (pm_message_t state);
```

## Arguments

*state* PM transition of the system being carried out.

## Description

Prepare all non-sysdev devices for system PM transition and execute “suspend” callbacks for them.

## Name

`device_pm_wait_for_dev` — Wait for suspend/resume of a device to complete.

## Synopsis

```
int device_pm_wait_for_dev (struct device * subordinate, struct device  
* dev);
```

## Arguments

*subordinate*    Device that needs to wait for *dev*.

*dev*            Device to wait for.

## Name

`dpm_for_each_dev` — device iterator.

## Synopsis

```
void dpm_for_each_dev (void * data, void (*fn) (struct device *, void  
*));
```

## Arguments

*data*    data for the callback.

*fn*      function to be called for each device.

## Description

Iterate over devices in `dpm_list`, and call *fn* for each device, passing it *data*.

# Device Drivers ACPI Support

## Name

`acpi_bus_scan` — Add ACPI device node objects in a given namespace scope.

## Synopsis

```
int acpi_bus_scan (acpi_handle handle);
```

## Arguments

*handle*    Root of the namespace scope to scan.

## Description

Scan a given ACPI tree (probably recently hot-plugged) and create and add found devices.

If no devices were found, `-ENODEV` is returned, but it does not mean that there has been a real error. There just have been no suitable ACPI objects in the table trunk from which the kernel could create a device and add an appropriate driver.

Must be called under `acpi_scan_lock`.

## Name

`acpi_bus_trim` — Detach scan handlers and drivers from ACPI device objects.

## Synopsis

```
void acpi_bus_trim (struct acpi_device * adev);
```

## Arguments

*adev*   Root of the ACPI namespace scope to walk.

## Description

Must be called under `acpi_scan_lock`.

## Name

`acpi_scan_drop_device` — Drop an ACPI device object.

## Synopsis

```
void acpi_scan_drop_device (acpi_handle handle, void * context);
```

## Arguments

*handle*     Handle of an ACPI namespace node, not used.

*context*   Address of the ACPI device object to drop.

## Description

This is invoked by `acpi_ns_delete_node` during the removal of the ACPI namespace node the device object pointed to by *context* is attached to.

The unregistration is carried out asynchronously to avoid running `acpi_device_del` under the ACPICA's namespace mutex and the list is used to ensure the correct ordering (the device objects must be unregistered in the same order in which the corresponding namespace nodes are deleted).

## Name

`acpi_dma_supported` — Check DMA support for the specified device.

## Synopsis

```
bool acpi_dma_supported (struct acpi_device * adev);
```

## Arguments

*adev*    The pointer to acpi device

## Description

Return false if DMA is not supported. Otherwise, return true



## Name

`acpi_get_dma_attr` — Check the supported DMA attr for the specified device.

## Synopsis

```
enum dev_dma_attr acpi_get_dma_attr (struct acpi_device * adev);
```

## Arguments

*adev*    The pointer to acpi device

## Description

Return enum `dev_dma_attr`.

# Device drivers PnP support

## Name

`pnp_register_protocol` — adds a pnp protocol to the pnp layer

## Synopsis

```
int pnp_register_protocol (struct pnp_protocol * protocol);
```

## Arguments

*protocol* pointer to the corresponding pnp\_protocol structure

## Ex protocols

ISAPNP, PNPBIOS, etc

## Name

`pnp_unregister_protocol` — removes a pnp protocol from the pnp layer

## Synopsis

```
void pnp_unregister_protocol (struct pnp_protocol * protocol);
```

## Arguments

*protocol* pointer to the corresponding `pnp_protocol` structure

## Name

`pnp_request_card_device` — Searches for a PnP device under the specified card

## Synopsis

```
struct pnp_dev * pnp_request_card_device (struct pnp_card_link * clink,  
const char * id, struct pnp_dev * from);
```

## Arguments

*clink*    pointer to the card link, cannot be NULL

*id*       pointer to a PnP ID structure that explains the rules for finding the device

*from*     Starting place to search from. If NULL it will start from the beginning.

## Name

`pnp_release_card_device` — call this when the driver no longer needs the device

## Synopsis

```
void pnp_release_card_device (struct pnp_dev * dev);
```

## Arguments

*dev* pointer to the PnP device structure

## Name

`pnp_register_card_driver` — registers a PnP card driver with the PnP Layer

## Synopsis

```
int pnp_register_card_driver (struct pnp_card_driver * drv);
```

## Arguments

*drv* pointer to the driver to register

## Name

`pnp_unregister_card_driver` — unregisters a PnP card driver from the PnP Layer

## Synopsis

```
void pnp_unregister_card_driver (struct pnp_card_driver * drv);
```

## Arguments

*drv* pointer to the driver to unregister

## Name

`pnp_add_id` — adds an EISA id to the specified device

## Synopsis

```
struct pnp_id * pnp_add_id (struct pnp_dev * dev, const char * id);
```

## Arguments

*dev* pointer to the desired device

*id* pointer to an EISA id string



## Name

`pnp_start_dev` — low-level start of the PnP device

## Synopsis

```
int pnp_start_dev (struct pnp_dev * dev);
```

## Arguments

*dev* pointer to the desired device

## Description

assumes that resources have already been allocated

## Name

`pnp_stop_dev` — low-level disable of the PnP device

## Synopsis

```
int pnp_stop_dev (struct pnp_dev * dev);
```

## Arguments

*dev* pointer to the desired device

## Description

does not free resources

## Name

`pnp_activate_dev` — activates a PnP device for use

## Synopsis

```
int pnp_activate_dev (struct pnp_dev * dev);
```

## Arguments

*dev* pointer to the desired device

## Description

does not validate or set resources so be careful.

## Name

`pnp_disable_dev` — disables device

## Synopsis

```
int pnp_disable_dev (struct pnp_dev * dev);
```

## Arguments

*dev* pointer to the desired device

## Description

inform the correct pnp protocol so that resources can be used by other devices

## Name

`pnp_is_active` — Determines if a device is active based on its current resources

## Synopsis

```
int pnp_is_active (struct pnp_dev * dev);
```

## Arguments

*dev* pointer to the desired PnP device

## Userspace IO devices

## Name

`uio_event_notify` — trigger an interrupt event

## Synopsis

```
void uio_event_notify (struct uio_info * info);
```

## Arguments

*info*    UIO device capabilities

## Name

`__uio_register_device` — register a new userspace IO device

## Synopsis

```
int __uio_register_device (struct module * owner, struct device * parent,  
struct uio_info * info);
```

## Arguments

*owner*     module that creates the new device

*parent*   parent device

*info*     UIO device capabilities

## Description

returns zero on success or a negative error code.

## Name

`uio_unregister_device` — unregister a industrial IO device

## Synopsis

```
void uio_unregister_device (struct uio_info * info);
```

## Arguments

*info*   UIO device capabilities



## Name

struct uio\_mem — description of a UIO memory region

## Synopsis

```
struct uio_mem {
    const char * name;
    phys_addr_t addr;
    resource_size_t size;
    int memtype;
    void __iomem * internal_addr;
    struct uio_map * map;
};
```

## Members

name	name of the memory region for identification
addr	address of the device's memory (phys_addr is used since addr can be logical, virtual, or physical & phys_addr_t should always be large enough to handle any of the address types)
size	size of IO
memtype	type of memory addr points to
internal_addr	ioremap-ped version of addr, for driver internal use
map	for use by the UIO core only.

## Name

struct uio\_port — description of a UIO port region

## Synopsis

```
struct uio_port {  
    const char * name;  
    unsigned long start;  
    unsigned long size;  
    int porttype;  
    struct uio_portio * portio;  
};
```

## Members

name	name of the port region for identification
start	start of port region
size	size of port region
porttype	type of port (see UIO_PORT_* below)
portio	for use by the UIO core only.

## Name

struct uio\_info — UIO device capabilities

## Synopsis

```
struct uio_info {
    struct uio_device * uio_dev;
    const char * name;
    const char * version;
    struct uio_mem mem[MAX_UIO_MAPS];
    struct uio_port port[MAX_UIO_PORT_REGIONS];
    long irq;
    unsigned long irq_flags;
    void * priv;
    irqreturn_t (* handler) (int irq, struct uio_info *dev_info);
    int (* mmap) (struct uio_info *info, struct vm_area_struct *vma);
    int (* open) (struct uio_info *info, struct inode *inode);
    int (* release) (struct uio_info *info, struct inode *inode);
    int (* irqcontrol) (struct uio_info *info, s32 irq_on);
};
```

## Members

uio_dev	the UIO device this info belongs to
name	device name
version	device driver version
mem[MAX_UIO_MAPS]	list of mappable memory regions, size==0 for end of list
port[MAX_UIO_PORT_REGIONS]	list of port regions, size==0 for end of list
irq	interrupt number or UIO_IRQ_CUSTOM
irq_flags	flags for request_irq
priv	optional private data
handler	the device's irq handler
mmap	mmap operation for this uio device
open	open operation for this uio device
release	release operation for this uio device
irqcontrol	disable/enable irqs when 0/1 is written to /dev/uioX

---

## Chapter 3. Parallel Port Devices

## Name

`parport_yield` — relinquish a parallel port temporarily

## Synopsis

```
int parport_yield (struct pardevice * dev);
```

## Arguments

*dev*    a device on the parallel port

## Description

This function relinquishes the port if it would be helpful to other drivers to do so. Afterwards it tries to reclaim the port using `parport_claim`, and the return value is the same as for `parport_claim`. If it fails, the port is left unclaimed and it is the driver's responsibility to reclaim the port.

The `parport_yield` and `parport_yield_blocking` functions are for marking points in the driver at which other drivers may claim the port and use their devices. Yielding the port is similar to releasing it and reclaiming it, but is more efficient because no action is taken if there are no other devices needing the port. In fact, nothing is done even if there are other devices waiting but the current device is still within its “timeslice”. The default timeslice is half a second, but it can be adjusted via the `/proc` interface.

## Name

`parport_yield_blocking` — relinquish a parallel port temporarily

## Synopsis

```
int parport_yield_blocking (struct pardevice * dev);
```

## Arguments

*dev*    a device on the parallel port

## Description

This function relinquishes the port if it would be helpful to other drivers to do so. Afterwards it tries to reclaim the port using `parport_claim_or_block`, and the return value is the same as for `parport_claim_or_block`.

## Name

`parport_wait_event` — wait for an event on a parallel port

## Synopsis

```
int parport_wait_event (struct parport * port, signed long timeout);
```

## Arguments

*port*        port to wait on

*timeout*    time to wait (in jiffies)

## Description

This function waits for up to *timeout* jiffies for an interrupt to occur on a parallel port. If the port timeout is set to zero, it returns immediately.

If an interrupt occurs before the timeout period elapses, this function returns zero immediately. If it times out, it returns one. An error code less than zero indicates an error (most likely a pending signal), and the calling code should finish what it's doing as soon as it can.

## Name

`parport_wait_peripheral` — wait for status lines to change in 35ms

## Synopsis

```
int parport_wait_peripheral (struct parport * port, unsigned char mask,
unsigned char result);
```

## Arguments

*port*      port to watch

*mask*      status lines to watch

*result*    desired values of chosen status lines

## Description

This function waits until the masked status lines have the desired values, or until 35ms have elapsed (see IEEE 1284-1994 page 24 to 25 for why this value in particular is hardcoded). The *mask* and *result* parameters are bitmasks, with the bits defined by the constants in `parport.h`: `PARPORT_STATUS_BUSY`, and so on.

The port is polled quickly to start off with, in anticipation of a fast response from the peripheral. This fast polling time is configurable (using `/proc`), and defaults to 500usec. If the timeout for this port (see `parport_set_timeout`) is zero, the fast polling time is 35ms, and this function does not call `schedule`.

If the timeout for this port is non-zero, after the fast polling fails it uses `parport_wait_event` to wait for up to 10ms, waking up if an interrupt occurs.



## Name

`parport_negotiate` — negotiate an IEEE 1284 mode

## Synopsis

```
int parport_negotiate (struct parport * port, int mode);
```

## Arguments

*port*    port to use

*mode*    mode to negotiate to

## Description

Use this to negotiate to a particular IEEE 1284 transfer mode. The *mode* parameter should be one of the constants in `parport.h` starting `IEEE1284_MODE_`xxx.

The return value is 0 if the peripheral has accepted the negotiation to the mode specified, -1 if the peripheral is not IEEE 1284 compliant (or not present), or 1 if the peripheral has rejected the negotiation.

## Name

`parport_write` — write a block of data to a parallel port

## Synopsis

```
ssize_t parport_write (struct parport * port, const void * buffer,  
size_t len);
```

## Arguments

*port*      port to write to

*buffer*   data buffer (in kernel space)

*len*       number of bytes of data to transfer

## Description

This will write up to *len* bytes of *buffer* to the port specified, using the IEEE 1284 transfer mode most recently negotiated to (using `parport_negotiate`), as long as that mode supports forward transfers (host to peripheral).

It is the caller's responsibility to ensure that the first *len* bytes of *buffer* are valid.

This function returns the number of bytes transferred (if zero or positive), or else an error code.

## Name

`parport_read` — read a block of data from a parallel port

## Synopsis

```
ssize_t parport_read (struct parport * port, void * buffer, size_t len);
```

## Arguments

*port*      port to read from

*buffer*   data buffer (in kernel space)

*len*       number of bytes of data to transfer

## Description

This will read up to *len* bytes of *buffer* to the port specified, using the IEEE 1284 transfer mode most recently negotiated to (using `parport_negotiate`), as long as that mode supports reverse transfers (peripheral to host).

It is the caller's responsibility to ensure that the first *len* bytes of *buffer* are available to write to.

This function returns the number of bytes transferred (if zero or positive), or else an error code.

## Name

`parport_set_timeout` — set the inactivity timeout for a device

## Synopsis

```
long parport_set_timeout (struct pardevice * dev, long inactivity);
```

## Arguments

*dev*                device on a port

*inactivity*       inactivity timeout (in jiffies)

## Description

This sets the inactivity timeout for a particular device on a port. This affects functions like `parport_wait_peripheral`. The special value 0 means not to call `schedule` while dealing with this device.

The return value is the previous inactivity timeout.

Any callers of `parport_wait_event` for this device are woken up.

## Name

`__parport_register_driver` — register a parallel port device driver

## Synopsis

```
int __parport_register_driver (struct parport_driver * drv, struct
module * owner, const char * mod_name);
```

## Arguments

*drv*            structure describing the driver

*owner*        owner module of *drv*

*mod\_name*    module name string

## Description

This can be called by a parallel port device driver in order to receive notifications about ports being found in the system, as well as ports no longer available.

If `devmodel` is true then the new device model is used for registration.

The *drv* structure is allocated by the caller and must not be deallocated until after calling `parport_unregister_driver`.

## If using the non device model

The driver's `attach` function may block. The port that `attach` is given will be valid for the duration of the callback, but if the driver wants to take a copy of the pointer it must call `parport_get_port` to do so. Calling `parport_register_device` on that port will do this for you.

The driver's `detach` function may block. The port that `detach` is given will be valid for the duration of the callback, but if the driver wants to take a copy of the pointer it must call `parport_get_port` to do so.

Returns 0 on success. The non device model will always succeed, but the new device model can fail and will return the error code.

## Name

`parport_unregister_driver` — deregister a parallel port device driver

## Synopsis

```
void parport_unregister_driver (struct parport_driver * drv);
```

## Arguments

*drv* structure describing the driver that was given to `parport_register_driver`

## Description

This should be called by a parallel port device driver that has registered itself using `parport_register_driver` when it is about to be unloaded.

When it returns, the driver's `attach` routine will no longer be called, and for each port that `attach` was called for, the `detach` routine will have been called.

All the driver's `attach` and `detach` calls are guaranteed to have finished by the time this function returns.

## Name

`parport_get_port` — increment a port's reference count

## Synopsis

```
struct parport * parport_get_port (struct parport * port);
```

## Arguments

*port* the port

## Description

This ensures that a struct parport pointer remains valid until the matching `parport_put_port` call.

## Name

`parport_put_port` — decrement a port's reference count

## Synopsis

```
void parport_put_port (struct parport * port);
```

## Arguments

*port*    the port

## Description

This should be called once for each call to `parport_get_port`, once the port is no longer needed. When the reference count reaches zero (port is no longer used), `free_port` is called.



## Name

`parport_register_port` — register a parallel port

## Synopsis

```
struct parport * parport_register_port (unsigned long base, int irq,  
int dma, struct parport_operations * ops);
```

## Arguments

*base*    base I/O address

*irq*     IRQ line

*dma*     DMA channel

*ops*     pointer to the port driver's port operations structure

## Description

When a parallel port (lowlevel) driver finds a port that should be made available to parallel port device drivers, it should call `parport_register_port`. The *base*, *irq*, and *dma* parameters are for the convenience of port drivers, and for ports where they aren't meaningful needn't be set to anything special. They can be altered afterwards by adjusting the relevant members of the `parport` structure that is returned and represents the port. They should not be tampered with after calling `parport_announce_port`, however.

If there are parallel port device drivers in the system that have registered themselves using `parport_register_driver`, they are not told about the port at this time; that is done by `parport_announce_port`.

The *ops* structure is allocated by the caller, and must not be deallocated before calling `parport_remove_port`.

If there is no memory to allocate a new `parport` structure, this function will return `NULL`.

## Name

`parport_announce_port` — tell device drivers about a parallel port

## Synopsis

```
void parport_announce_port (struct parport * port);
```

## Arguments

*port* parallel port to announce

## Description

After a port driver has registered a parallel port with `parport_register_port`, and performed any necessary initialisation or adjustments, it should call `parport_announce_port` in order to notify all device drivers that have called `parport_register_driver`. Their attach functions will be called, with *port* as the parameter.

## Name

`parport_remove_port` — deregister a parallel port

## Synopsis

```
void parport_remove_port (struct parport * port);
```

## Arguments

*port* parallel port to deregister

## Description

When a parallel port driver is forcibly unloaded, or a parallel port becomes inaccessible, the port driver must call this function in order to deal with device drivers that still want to use it.

The `parport` structure associated with the port has its operations structure replaced with one containing 'null' operations that return errors or just don't do anything.

Any drivers that have registered themselves using `parport_register_driver` are notified that the port is no longer accessible by having their `detach` routines called with *port* as the parameter.

## Name

`parport_register_device` — register a device on a parallel port

## Synopsis

```
struct pardevice * parport_register_device (struct parport * port, const
char * name, int (*pf) (void *), void (*kf) (void *), void (*irq_func)
(void *), int flags, void * handle);
```

## Arguments

<i>port</i>	port to which the device is attached
<i>name</i>	a name to refer to the device
<i>pf</i>	preemption callback
<i>kf</i>	kick callback (wake-up)
<i>irq_func</i>	interrupt handler
<i>flags</i>	registration flags
<i>handle</i>	data for callback functions

## Description

This function, called by parallel port device drivers, declares that a device is connected to a port, and tells the system all it needs to know.

The *name* is allocated by the caller and must not be deallocated until the caller calls `parport_unregister_device` for that device.

The preemption callback function, *pf*, is called when this device driver has claimed access to the port but another device driver wants to use it. It is given *handle* as its parameter, and should return zero if it is willing for the system to release the port to another driver on its behalf. If it wants to keep control of the port it should return non-zero, and no action will be taken. It is good manners for the driver to try to release the port at the earliest opportunity after its preemption callback rejects a preemption attempt. Note that if a preemption callback is happy for preemption to go ahead, there is no need to release the port; it is done automatically. This function may not block, as it may be called from interrupt context. If the device driver does not support preemption, *pf* can be NULL.

The wake-up (“kick”) callback function, *kf*, is called when the port is available to be claimed for exclusive access; that is, `parport_claim` is guaranteed to succeed when called from inside the wake-up callback function. If the driver wants to claim the port it should do so; otherwise, it need not take any action. This function may not block, as it may be called from interrupt context. If the device driver does not want to be explicitly invited to claim the port in this way, *kf* can be NULL.

The interrupt handler, *irq\_func*, is called when an interrupt arrives from the parallel port. Note that if a device driver wants to use interrupts it should use `parport_enable_irq`, and can also check the `irq` member of the `parport` structure representing the port.

The parallel port (lowlevel) driver is the one that has called `request_irq` and whose interrupt handler is called first. This handler does whatever needs to be done to the hardware to acknowledge the interrupt

(for PC-style ports there is nothing special to be done). It then tells the IEEE 1284 code about the interrupt, which may involve reacting to an IEEE 1284 event depending on the current IEEE 1284 phase. After this, it calls *irq\_func*. Needless to say, *irq\_func* will be called from interrupt context, and may not block.

The `PARPORT_DEV_EXCL` flag is for preventing port sharing, and so should only be used when sharing the port with other device drivers is impossible and would lead to incorrect behaviour. Use it sparingly! Normally, *flags* will be zero.

This function returns a pointer to a structure that represents the device on the port, or `NULL` if there is not enough memory to allocate space for that structure.

## Name

`parport_unregister_device` — deregister a device on a parallel port

## Synopsis

```
void parport_unregister_device (struct pardevice * dev);
```

## Arguments

*dev* pointer to structure representing device

## Description

This undoes the effect of `parport_register_device`.

## Name

`parport_find_number` — find a parallel port by number

## Synopsis

```
struct parport * parport_find_number (int number);
```

## Arguments

*number* parallel port number

## Description

This returns the parallel port with the specified number, or `NULL` if there is none.

There is an implicit `parport_get_port` done already; to throw away the reference to the port that `parport_find_number` gives you, use `parport_put_port`.

## Name

`parport_find_base` — find a parallel port by base address

## Synopsis

```
struct parport * parport_find_base (unsigned long base);
```

## Arguments

*base*    base I/O address

## Description

This returns the parallel port with the specified base address, or NULL if there is none.

There is an implicit `parport_get_port` done already; to throw away the reference to the port that `parport_find_base` gives you, use `parport_put_port`.



## Name

`parport_claim` — claim access to a parallel port device

## Synopsis

```
int parport_claim (struct pardevice * dev);
```

## Arguments

*dev* pointer to structure representing a device on the port

## Description

This function will not block and so can be used from interrupt context. If `parport_claim` succeeds in claiming access to the port it returns zero and the port is available to use. It may fail (returning non-zero) if the port is in use by another driver and that driver is not willing to relinquish control of the port.

## Name

`parport_claim_or_block` — claim access to a parallel port device

## Synopsis

```
int parport_claim_or_block (struct pardevice * dev);
```

## Arguments

*dev* pointer to structure representing a device on the port

## Description

This behaves like `parport_claim`, but will block if necessary to wait for the port to be free. A return value of 1 indicates that it slept; 0 means that it succeeded without needing to sleep. A negative error code indicates failure.

## Name

`parport_release` — give up access to a parallel port device

## Synopsis

```
void parport_release (struct pardevice * dev);
```

## Arguments

*dev* pointer to structure representing parallel port device

## Description

This function cannot fail, but it should not be called without the port claimed. Similarly, if the port is already claimed you should not try claiming it again.

## Name

`parport_open` — find a device by canonical device number

## Synopsis

```
struct pardevice * parport_open (int devnum, const char * name);
```

## Arguments

*devnum*    canonical device number

*name*      name to associate with the device

## Description

This function is similar to `parport_register_device`, except that it locates a device by its number rather than by the port it is attached to.

All parameters except for *devnum* are the same as for `parport_register_device`. The return value is the same as for `parport_register_device`.

## Name

`parport_close` — close a device opened with `parport_open`

## Synopsis

```
void parport_close (struct pardevice * dev);
```

## Arguments

*dev*    device to close

## Description

This is to `parport_open` as `parport_unregister_device` is to `parport_register_device`.

---

# **Chapter 4. Message-based devices**

## **Fusion message devices**

## Name

`mpt_register` — Register protocol-specific main callback handler.

## Synopsis

```
u8 mpt_register (MPT_CALLBACK cbfunc, MPT_DRIVER_CLASS dclass, char *  
func_name) ;
```

## Arguments

<i>cbfunc</i>	callback function pointer
<i>dclass</i>	Protocol driver's class (MPT_DRIVER_CLASS enum value)
<i>func_name</i>	call function's name

## Description

This routine is called by a protocol-specific driver (SCSI host, LAN, SCSI target) to register its reply callback routine. Each protocol-specific driver must do this before it will be able to use any IOC resources, such as obtaining request frames.

## NOTES

The SCSI protocol driver currently calls this routine thrice in order to register separate callbacks; one for “normal” SCSI IO; one for `MptScsiTaskMgmt` requests; one for Scan/DV requests.

Returns u8 valued “handle” in the range (and S.O.D. order) {N,...,7,6,5,...,1} if successful. A return value of `MPT_MAX_PROTOCOL_DRIVERS` (including zero!) should be considered an error by the caller.

## Name

`mpt_deregister` — Deregister a protocol drivers resources.

## Synopsis

```
void mpt_deregister (u8 cb_idx);
```

## Arguments

*cb\_idx*    previously registered callback handle

## Description

Each protocol-specific driver should call this routine when its module is unloaded.



## Name

`mpt_event_register` — Register protocol-specific event callback handler.

## Synopsis

```
int mpt_event_register (u8 cb_idx, MPT_EVHANDLER ev_cbfunc);
```

## Arguments

*cb\_idx*            previously registered (via `mpt_register`) callback handle

*ev\_cbfunc*        callback function

## Description

This routine can be called by one or more protocol-specific drivers if/when they choose to be notified of MPT events.

Returns 0 for success.

## Name

`mpt_event_deregister` — Deregister protocol-specific event callback handler

## Synopsis

```
void mpt_event_deregister (u8 cb_idx);
```

## Arguments

*cb\_idx*    previously registered callback handle

## Description

Each protocol-specific driver should call this routine when it does not (or can no longer) handle events, or when its module is unloaded.

## Name

`mpt_reset_register` — Register protocol-specific IOC reset handler.

## Synopsis

```
int mpt_reset_register (u8 cb_idx, MPT_RESETHANDLER reset_func);
```

## Arguments

*cb\_idx*            previously registered (via `mpt_register`) callback handle

*reset\_func*    reset function

## Description

This routine can be called by one or more protocol-specific drivers if/when they choose to be notified of IOC resets.

Returns 0 for success.

## Name

`mpt_reset_deregister` — Deregister protocol-specific IOC reset handler.

## Synopsis

```
void mpt_reset_deregister (u8 cb_idx);
```

## Arguments

*cb\_idx*    previously registered callback handle

## Description

Each protocol-specific driver should call this routine when it does not (or can no longer) handle IOC reset handling, or when its module is unloaded.

## Name

`mpt_device_driver_register` — Register device driver hooks

## Synopsis

```
int mpt_device_driver_register (struct mpt_pci_driver * dd_cbfunc, u8  
cb_idx);
```

## Arguments

*dd\_cbfunc*    driver callbacks struct

*cb\_idx*       MPT protocol driver index

## Name

mpt\_device\_driver\_deregister — DeRegister device driver hooks

## Synopsis

```
void mpt_device_driver_deregister (u8 cb_idx);
```

## Arguments

*cb\_idx* MPT protocol driver index

## Name

`mpt_get_msg_frame` — Obtain an MPT request frame from the pool

## Synopsis

```
MPT_FRAME_HDR* mpt_get_msg_frame (u8 cb_idx, MPT_ADAPTER * ioc);
```

## Arguments

*cb\_idx*     Handle of registered MPT protocol driver

*ioc*        Pointer to MPT adapter structure

## Description

Obtain an MPT request frame from the pool (of 1024) that are allocated per MPT adapter.

Returns pointer to a MPT request frame or NULL if none are available or IOC is not active.

## Name

`mpt_put_msg_frame` — Send a protocol-specific MPT request frame to an IOC

## Synopsis

```
void mpt_put_msg_frame (u8 cb_idx, MPT_ADAPTER * ioc, MPT_FRAME_HDR *  
mf);
```

## Arguments

*cb\_idx*    Handle of registered MPT protocol driver

*ioc*        Pointer to MPT adapter structure

*mf*        Pointer to MPT request frame

## Description

This routine posts an MPT request frame to the request post FIFO of a specific MPT adapter.



## Name

`mpt_put_msg_frame_hi_pri` — Send a hi-pri protocol-specific MPT request frame

## Synopsis

```
void    mpt_put_msg_frame_hi_pri    (u8    cb_idx,    MPT_ADAPTER    *    ioc,  
MPT_FRAME_HDR    *    mf);
```

## Arguments

*cb\_idx* Handle of registered MPT protocol driver

*ioc* Pointer to MPT adapter structure

*mf* Pointer to MPT request frame

## Description

Send a protocol-specific MPT request frame to an IOC using hi-priority request queue.

This routine posts an MPT request frame to the request post FIFO of a specific MPT adapter.

## Name

`mpt_free_msg_frame` — Place MPT request frame back on FreeQ.

## Synopsis

```
void mpt_free_msg_frame (MPT_ADAPTER * ioc, MPT_FRAME_HDR * mf);
```

## Arguments

*ioc*    Pointer to MPT adapter structure

*mf*     Pointer to MPT request frame

## Description

This routine places a MPT request frame back on the MPT adapter's FreeQ.

## Name

`mpt_send_handshake_request` — Send MPT request via doorbell handshake method.

## Synopsis

```
int mpt_send_handshake_request (u8  cb_idx, MPT_ADAPTER * ioc, int
reqBytes, u32 * req, int sleepFlag);
```

## Arguments

<i>cb_idx</i>	Handle of registered MPT protocol driver
<i>ioc</i>	Pointer to MPT adapter structure
<i>reqBytes</i>	Size of the request in bytes
<i>req</i>	Pointer to MPT request frame
<i>sleepFlag</i>	Use schedule if CAN_SLEEP else use udelay.

## Description

This routine is used exclusively to send `MptScsiTaskMgmt` requests since they are required to be sent via doorbell handshake.

## NOTE

It is the callers responsibility to byte-swap fields in the request which are greater than 1 byte in size.

Returns 0 for success, non-zero for failure.

## Name

`mpt_verify_adapter` — Given IOC identifier, set pointer to its adapter structure.

## Synopsis

```
int mpt_verify_adapter (int iocid, MPT_ADAPTER ** iocpp);
```

## Arguments

*iocid*    IOC unique identifier (integer)

*iocpp*    Pointer to pointer to IOC adapter

## Description

Given a unique IOC identifier, set pointer to the associated MPT adapter structure.

Returns `iocid` and sets `iocpp` if `iocid` is found. Returns -1 if `iocid` is not found.

## Name

`mpt_attach` — Install a PCI intelligent MPT adapter.

## Synopsis

```
int mpt_attach (struct pci_dev * pdev, const struct pci_device_id * id);
```

## Arguments

*pdev*    Pointer to `pci_dev` structure

*id*       PCI device ID information

## Description

This routine performs all the steps necessary to bring the IOC of a MPT adapter to a OPERATIONAL state. This includes registering memory regions, registering the interrupt, and allocating request and reply memory pools.

This routine also pre-fetches the LAN MAC address of a Fibre Channel MPT adapter.

Returns 0 for success, non-zero for failure.

## TODO

Add support for polled controllers

## Name

`mpt_detach` — Remove a PCI intelligent MPT adapter.

## Synopsis

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

## Arguments

*pdev*    Pointer to `pci_dev` structure

## Name

`mpt_suspend` — Fusion MPT base driver suspend routine.

## Synopsis

```
int mpt_suspend (struct pci_dev * pdev, pm_message_t state);
```

## Arguments

*pdev*    Pointer to `pci_dev` structure

*state*   new state to enter

## Name

`mpt_resume` — Fusion MPT base driver resume routine.

## Synopsis

```
int mpt_resume (struct pci_dev * pdev);
```

## Arguments

*pdev*    Pointer to `pci_dev` structure



## Name

`mpt_GetIocState` — Get the current state of a MPT adapter.

## Synopsis

```
u32 mpt_GetIocState (MPT_ADAPTER * ioc, int cooked);
```

## Arguments

*ioc*      Pointer to MPT\_ADAPTER structure

*cooked*   Request raw or cooked IOC state

## Description

Returns all IOC Doorbell register bits if `cooked==0`, else just the Doorbell bits in `MPI_IOC_STATE_MASK`.

## Name

`mpt_alloc_fw_memory` — allocate firmware memory

## Synopsis

```
int mpt_alloc_fw_memory (MPT_ADAPTER * ioc, int size);
```

## Arguments

*ioc*    Pointer to MPT\_ADAPTER structure

*size*   total FW bytes

## Description

If memory has already been allocated, the same (cached) value is returned.

Return 0 if successful, or non-zero for failure

## Name

`mpt_free_fw_memory` — free firmware memory

## Synopsis

```
void mpt_free_fw_memory (MPT_ADAPTER * ioc);
```

## Arguments

*ioc* Pointer to MPT\_ADAPTER structure

## Description

If `alt_img` is NULL, delete from `ioc` structure. Else, delete a secondary image in same format.

## Name

`mptbase_sas_persist_operation` — Perform operation on SAS Persistent Table

## Synopsis

```
int    mptbase_sas_persist_operation    (MPT_ADAPTER    *    ioc,    u8
persist_opcode);
```

## Arguments

*ioc*                      Pointer to MPT\_ADAPTER structure

*persist\_opcode*    see below

## Description

MPI\_SAS\_OP\_CLEAR\_NOT\_PRESENT - Free all persist TargetID mappings for devices not currently present. MPI\_SAS\_OP\_CLEAR\_ALL\_PERSISTENT - Clear al persist TargetID mappings

## NOTE

Don't use not this function during interrupt time.

Returns 0 for success, non-zero error

## Name

`mpt_raid_phys_disk_pg0` — returns phys disk page zero

## Synopsis

```
int mpt_raid_phys_disk_pg0 (MPT_ADAPTER * ioc, u8 phys_disk_num,
    RaidPhysDiskPage0_t * phys_disk);
```

## Arguments

<i>ioc</i>	Pointer to a Adapter Structure
<i>phys_disk_num</i>	io unit unique phys disk num generated by the ioc
<i>phys_disk</i>	requested payload data returned

## Return

0 on success -EFAULT if read of config page header fails or data pointer not NULL -ENOMEM if pci\_alloc failed

## Name

`mpt_raid_phys_disk_get_num_paths` — returns number paths associated to this `phys_num`

## Synopsis

```
int    mpt_raid_phys_disk_get_num_paths    (MPT_ADAPTER    *    ioc,    u8
phys_disk_num);
```

## Arguments

*ioc*                      Pointer to a Adapter Structure

*phys\_disk\_num*    io unit unique phys disk num generated by the ioc

## Return

returns number paths

## Name

`mpt_raid_phys_disk_pg1` — returns phys disk page 1

## Synopsis

```
int mpt_raid_phys_disk_pg1 (MPT_ADAPTER * ioc, u8 phys_disk_num,
    RaidPhysDiskPage1_t * phys_disk);
```

## Arguments

<i>ioc</i>	Pointer to a Adapter Structure
<i>phys_disk_num</i>	io unit unique phys disk num generated by the ioc
<i>phys_disk</i>	requested payload data returned

## Return

0 on success -EFAULT if read of config page header fails or data pointer not NULL -ENOMEM if pci\_alloc failed

## Name

`mpt_findImVolumes` — Identify IDs of hidden disks and RAID Volumes

## Synopsis

```
int mpt_findImVolumes (MPT_ADAPTER * ioc);
```

## Arguments

*ioc* Pointer to a Adapter Strucutre

## Return

0 on success -EFAULT if read of config page header fails or data pointer not NULL -ENOMEM if pci\_alloc failed



## Name

`mpt_config` — Generic function to issue config message

## Synopsis

```
int mpt_config (MPT_ADAPTER * ioc, CONFIGPARMS * pCfg);
```

## Arguments

*ioc*     Pointer to an adapter structure

*pCfg*   Pointer to a configuration structure. Struct contains action, page address, direction, physical address and pointer to a configuration page header Page header is updated.

## Description

Returns 0 for success -EPERM if not allowed due to ISR context -EAGAIN if no msg frames currently available -EFAULT for non-successful reply or no reply (timeout)

## Name

`mpt_print_ioc_summary` — Write ASCII summary of IOC to a buffer.

## Synopsis

```
void mpt_print_ioc_summary (MPT_ADAPTER * ioc, char * buffer, int *  
size, int len, int showlan);
```

## Arguments

<i>ioc</i>	Pointer to MPT_ADAPTER structure
<i>buffer</i>	Pointer to buffer where IOC summary info should be written
<i>size</i>	Pointer to number of bytes we wrote (set by this routine)
<i>len</i>	Offset at which to start writing in buffer
<i>showlan</i>	Display LAN stuff?

## Description

This routine writes (english readable) ASCII text, which represents a summary of IOC information, to a buffer.

## Name

`mpt_set_taskmgmt_in_progress_flag` — set flags associated with task management

## Synopsis

```
int mpt_set_taskmgmt_in_progress_flag (MPT_ADAPTER * ioc);
```

## Arguments

*ioc* Pointer to MPT\_ADAPTER structure

## Description

Returns 0 for SUCCESS or -1 if FAILED.

If -1 is return, then it was not possible to set the flags

## Name

`mpt_clear_taskmgmt_in_progress_flag` — clear flags associated with task management

## Synopsis

```
void mpt_clear_taskmgmt_in_progress_flag (MPT_ADAPTER * ioc);
```

## Arguments

*ioc* Pointer to MPT\_ADAPTER structure

## Name

`mpt_halt_firmware` — Halts the firmware if it is operational and panic the kernel

## Synopsis

```
void mpt_halt_firmware (MPT_ADAPTER * ioc);
```

## Arguments

*ioc*    Pointer to MPT\_ADAPTER structure

## Name

`mpt_Soft_Hard_ResetHandler` — Try less expensive reset

## Synopsis

```
int mpt_Soft_Hard_ResetHandler (MPT_ADAPTER * ioc, int sleepFlag);
```

## Arguments

*ioc*            Pointer to MPT\_ADAPTER structure

*sleepFlag*   Indicates if sleep or schedule must be called.

## Description

Returns 0 for SUCCESS or -1 if FAILED. Try for softreset first, only if it fails go for expensive HardReset.

## Name

`mpt_HardResetHandler` — Generic reset handler

## Synopsis

```
int mpt_HardResetHandler (MPT_ADAPTER * ioc, int sleepFlag);
```

## Arguments

*ioc*                      Pointer to MPT\_ADAPTER structure

*sleepFlag*            Indicates if sleep or schedule must be called.

## Description

Issues SCSI Task Management call based on input arg values. If TaskMgmt fails, returns associated SCSI request.

## Remark

`_HardResetHandler` can be invoked from an interrupt thread (timer) or a non-interrupt thread. In the former, must not call `schedule`.

## Note

A return of -1 is a FATAL error case, as it means a FW reload/initialization failed.

Returns 0 for SUCCESS or -1 if FAILED.

## Name

`mpt_get_cb_idx` — obtain `cb_idx` for registered driver

## Synopsis

```
u8 mpt_get_cb_idx (MPT_DRIVER_CLASS dclass);
```

## Arguments

*dclass*    class driver enum

## Description

Returns `cb_idx`, or zero means it wasn't found



## Name

`mpt_is_discovery_complete` — determine if discovery has completed

## Synopsis

```
int mpt_is_discovery_complete (MPT_ADAPTER * ioc);
```

## Arguments

*ioc* per adapter instance

## Description

Returns 1 when discovery completed, else zero.

## Name

`mpt_remove_dead_ioc_func` — kthread context to remove dead ioc

## Synopsis

```
int mpt_remove_dead_ioc_func (void * arg);
```

## Arguments

*arg* input argument, used to derive ioc

## Description

Return 0 if controller is removed from pci subsystem. Return -1 for other case.

## Name

`mpt_fault_reset_work` — work performed on workq after ioc fault

## Synopsis

```
void mpt_fault_reset_work (struct work_struct * work);
```

## Arguments

*work* input argument, used to derive ioc

## Name

`mpt_interrupt` — MPT adapter (IOC) specific interrupt handler.

## Synopsis

```
irqreturn_t mpt_interrupt (int irq, void * bus_id);
```

## Arguments

*irq*        irq number (not used)

*bus\_id*    bus identifier cookie == pointer to MPT\_ADAPTER structure

## Description

This routine is registered via the `request_irq` kernel API call, and handles all interrupts generated from a specific MPT adapter (also referred to as a IO Controller or IOC). This routine must clear the interrupt from the adapter and does so by reading the reply FIFO. Multiple replies may be processed per single call to this routine.

This routine handles register-level access of the adapter but dispatches (calls) a protocol-specific callback routine to handle the protocol-specific details of the MPT request completion.

## Name

`mptbase_reply` — MPT base driver's callback routine

## Synopsis

```
int mptbase_reply (MPT_ADAPTER * ioc, MPT_FRAME_HDR * req, MPT_FRAME_HDR  
* reply);
```

## Arguments

*ioc*     Pointer to MPT\_ADAPTER structure

*req*     Pointer to original MPT request frame

*reply*   Pointer to MPT reply frame (NULL if TurboReply)

## Description

MPT base driver's callback routine; all base driver “internal” request/reply processing is routed here. Currently used for EventNotification and EventAck handling.

Returns 1 indicating original alloc'd request frame ptr should be freed, or 0 if it shouldn't.

## Name

`mpt_add_sge` — Place a simple 32 bit SGE at address `pAddr`.

## Synopsis

```
void mpt_add_sge (void * pAddr, u32 flagslength, dma_addr_t dma_addr);
```

## Arguments

<i>pAddr</i>	virtual address for SGE
<i>flagslength</i>	SGE flags and data transfer length
<i>dma_addr</i>	Physical address

## Description

This routine places a MPT request frame back on the MPT adapter's FreeQ.

## Name

`mpt_add_sge_64bit` — Place a simple 64 bit SGE at address `pAddr`.

## Synopsis

```
void mpt_add_sge_64bit (void * pAddr, u32 flagslength, dma_addr_t  
                        dma_addr);
```

## Arguments

<i>pAddr</i>	virtual address for SGE
<i>flagslength</i>	SGE flags and data transfer length
<i>dma_addr</i>	Physical address

## Description

This routine places a MPT request frame back on the MPT adapter's FreeQ.

## Name

`mpt_add_sge_64bit_1078` — Place a simple 64 bit SGE at address `pAddr` (1078 workaround).

## Synopsis

```
void mpt_add_sge_64bit_1078 (void * pAddr, u32 flagslength, dma_addr_t
dma_addr);
```

## Arguments

<i>pAddr</i>	virtual address for SGE
<i>flagslength</i>	SGE flags and data transfer length
<i>dma_addr</i>	Physical address

## Description

This routine places a MPT request frame back on the MPT adapter's FreeQ.



## Name

`mpt_add_chain` — Place a 32 bit chain SGE at address `pAddr`.

## Synopsis

```
void mpt_add_chain (void * pAddr, u8 next, u16 length, dma_addr_t
dma_addr);
```

## Arguments

<i>pAddr</i>	virtual address for SGE
<i>next</i>	nextChainOffset value (u32's)
<i>length</i>	length of next SGL segment
<i>dma_addr</i>	Physical address

## Name

`mpt_add_chain_64bit` — Place a 64 bit chain SGE at address `pAddr`.

## Synopsis

```
void mpt_add_chain_64bit (void * pAddr, u8 next, u16 length, dma_addr_t
dma_addr);
```

## Arguments

<i>pAddr</i>	virtual address for SGE
<i>next</i>	nextChainOffset value (u32's)
<i>length</i>	length of next SGL segment
<i>dma_addr</i>	Physical address

## Name

`mpt_host_page_access_control` — control the IOC's Host Page Buffer access

## Synopsis

```
int      mpt_host_page_access_control (MPT_ADAPTER * ioc, u8
access_control_value, int sleepFlag);
```

## Arguments

<i>ioc</i>	Pointer to MPT adapter structure
<i>access_control_value</i>	define bits below
<i>sleepFlag</i>	Specifies whether the process can sleep

## Description

Provides mechanism for the host driver to control the IOC's Host Page Buffer access.

Access	Control	Value	-	bits[15:12]	0h	Reserved	1h	Enable
Access	{	MPI_DB_HPBACK_ENABLE_ACCESS	}		2h	Disable		Access
	{	MPI_DB_HPBACK_DISABLE_ACCESS	}	3h	Free Buffer	{	MPI_DB_HPBACK_FREE_BUFFER	}

Returns 0 for success, non-zero for failure.

## Name

`mpt_host_page_alloc` — allocate system memory for the fw

## Synopsis

```
int mpt_host_page_alloc (MPT_ADAPTER * ioc, pIOCInit_t ioc_init);
```

## Arguments

*ioc*            Pointer to pointer to IOC adapter

*ioc\_init*      Pointer to ioc init config page

## Description

If we already allocated memory in past, then resend the same pointer. Returns 0 for success, non-zero for failure.

## Name

`mpt_get_product_name` — returns product string

## Synopsis

```
const char* mpt_get_product_name (u16 vendor, u16 device, u8 revision);
```

## Arguments

<i>vendor</i>	pci vendor id
<i>device</i>	pci device id
<i>revision</i>	pci revision id

## Description

Returns product string displayed when driver loads, in `/proc/mpt/summary` and `/sysfs/class/scsi_host/host<X>/version_product`

## Name

`mpt_mapresources` — map in memory mapped io

## Synopsis

```
int mpt_mapresources (MPT_ADAPTER * ioc);
```

## Arguments

*ioc*    Pointer to pointer to IOC adapter

## Name

`mpt_do_ioc_recovery` — Initialize or recover MPT adapter.

## Synopsis

```
int mpt_do_ioc_recovery (MPT_ADAPTER * ioc, u32 reason, int sleepFlag);
```

## Arguments

<i>ioc</i>	Pointer to MPT adapter structure
<i>reason</i>	Event word / reason
<i>sleepFlag</i>	Use schedule if CAN_SLEEP else use udelay.

## Description

This routine performs all the steps necessary to bring the IOC to a OPERATIONAL state.

This routine also pre-fetches the LAN MAC address of a Fibre Channel MPT adapter.

## Returns

0 for success -1 if failed to get board READY -2 if READY but IOCFacts Failed -3 if READY but PrimeIOCFifos Failed -4 if READY but IOCInit Failed -5 if failed to enable\_device and/or request\_selected\_regions -6 if failed to upload firmware

## Name

`mpt_detect_bound_ports` — Search for matching PCI bus/dev\_function

## Synopsis

```
void mpt_detect_bound_ports (MPT_ADAPTER * ioc, struct pci_dev * pdev);
```

## Arguments

*ioc*     Pointer to MPT adapter structure

*pdev*    Pointer to (struct pci\_dev) structure

## Description

Search for PCI bus/dev\_function which matches PCI bus/dev\_function (+/-1) for newly discovered 929, 929X, 1030 or 1035.

If match on PCI dev\_function +/-1 is found, bind the two MPT adapters using alt\_ioc pointer fields in their MPT\_ADAPTER structures.



## Name

`mpt_adapter_disable` — Disable misbehaving MPT adapter.

## Synopsis

```
void mpt_adapter_disable (MPT_ADAPTER * ioc);
```

## Arguments

*ioc*    Pointer to MPT adapter structure

## Name

`mpt_adapter_dispose` — Free all resources associated with an MPT adapter

## Synopsis

```
void mpt_adapter_dispose (MPT_ADAPTER * ioc);
```

## Arguments

*ioc* Pointer to MPT adapter structure

## Description

This routine unregisters h/w resources and frees all alloc'd memory associated with a MPT adapter structure.

## Name

MptDisplayIocCapabilities — Display IOC's capabilities.

## Synopsis

```
void MptDisplayIocCapabilities (MPT_ADAPTER * ioc);
```

## Arguments

*ioc* Pointer to MPT adapter structure

## Name

MakeIocReady — Get IOC to a READY state, using KickStart if needed.

## Synopsis

```
int MakeIocReady (MPT_ADAPTER * ioc, int force, int sleepFlag);
```

## Arguments

*ioc*            Pointer to MPT\_ADAPTER structure

*force*        Force hard KickStart of IOC

*sleepFlag*    Specifies whether the process can sleep

## Returns

1 - DIAG reset and READY 0 - READY initially OR soft reset and READY -1 - Any failure on KickStart  
 -2 - Msg Unit Reset Failed -3 - IO Unit Reset Failed -4 - IOC owned by a PEER

## Name

GetIocFacts — Send IOCFacts request to MPT adapter.

## Synopsis

```
int GetIocFacts (MPT_ADAPTER * ioc, int sleepFlag, int reason);
```

## Arguments

<i>ioc</i>	Pointer to MPT_ADAPTER structure
<i>sleepFlag</i>	Specifies whether the process can sleep
<i>reason</i>	If recovery, only update facts.

## Description

Returns 0 for success, non-zero for failure.

## Name

GetPortFacts — Send PortFacts request to MPT adapter.

## Synopsis

```
int GetPortFacts (MPT_ADAPTER * ioc, int portnum, int sleepFlag);
```

## Arguments

<i>ioc</i>	Pointer to MPT_ADAPTER structure
<i>portnum</i>	Port number
<i>sleepFlag</i>	Specifies whether the process can sleep

## Description

Returns 0 for success, non-zero for failure.

## Name

SendIocInit — Send IOInit request to MPT adapter.

## Synopsis

```
int SendIocInit (MPT_ADAPTER * ioc, int sleepFlag);
```

## Arguments

*ioc*            Pointer to MPT\_ADAPTER structure

*sleepFlag*    Specifies whether the process can sleep

## Description

Send IOInit followed by PortEnable to bring IOC to OPERATIONAL state.

Returns 0 for success, non-zero for failure.

## Name

SendPortEnable — Send PortEnable request to MPT adapter port.

## Synopsis

```
int SendPortEnable (MPT_ADAPTER * ioc, int portnum, int sleepFlag);
```

## Arguments

<i>ioc</i>	Pointer to MPT_ADAPTER structure
<i>portnum</i>	Port number to enable
<i>sleepFlag</i>	Specifies whether the process can sleep

## Description

Send PortEnable to bring IOC to OPERATIONAL state.

Returns 0 for success, non-zero for failure.



## Name

`mpt_do_upload` — Construct and Send FWUpload request to MPT adapter port.

## Synopsis

```
int mpt_do_upload (MPT_ADAPTER * ioc, int sleepFlag);
```

## Arguments

*ioc*                      Pointer to MPT\_ADAPTER structure

*sleepFlag*              Specifies whether the process can sleep

## Description

Returns 0 for success, >0 for handshake failure <0 for fw upload failure.

## Remark

If bound IOC and a successful FWUpload was performed on the bound IOC, the second image is discarded and memory is free'd. Both channels must upload to prevent IOC from running in degraded mode.

## Name

mpt\_downloadboot — DownloadBoot code

## Synopsis

```
int mpt_downloadboot (MPT_ADAPTER * ioc, MpiFwHeader_t * pFwHeader, int
sleepFlag);
```

## Arguments

*ioc*            Pointer to MPT\_ADAPTER structure

*pFwHeader*    Pointer to firmware header info

*sleepFlag*    Specifies whether the process can sleep

## Description

FwDownloadBoot requires Programmed IO access.

Returns 0 for success -1 FW Image size is 0 -2 No valid cached\_fw Pointer <0 for fw upload failure.

## Name

KickStart — Perform hard reset of MPT adapter.

## Synopsis

```
int KickStart (MPT_ADAPTER * ioc, int force, int sleepFlag);
```

## Arguments

<i>ioc</i>	Pointer to MPT_ADAPTER structure
<i>force</i>	Force hard reset
<i>sleepFlag</i>	Specifies whether the process can sleep

## Description

This routine places MPT adapter in diagnostic mode via the WriteSequence register, and then performs a hard reset of adapter via the Diagnostic register.

## Inputs

sleepflag - CAN\_SLEEP (non-interrupt thread) or NO\_SLEEP (interrupt thread, use mdelay) force - 1 if doorbell active, board fault state board operational, IOC\_RECOVERY or IOC\_BRINGUP and there is an alt\_ioc. 0 else

## Returns

1 - hard reset, READY 0 - no reset due to History bit, READY -1 - no reset due to History bit but not READY OR reset but failed to come READY -2 - no reset, could not enter DIAG mode -3 - reset but bad FW bit

## Name

`mpt_diag_reset` — Perform hard reset of the adapter.

## Synopsis

```
int mpt_diag_reset (MPT_ADAPTER * ioc, int ignore, int sleepFlag);
```

## Arguments

<i>ioc</i>	Pointer to MPT_ADAPTER structure
<i>ignore</i>	Set if to honor and clear to ignore the reset history bit
<i>sleepFlag</i>	CAN_SLEEP if called in a non-interrupt thread, else set to NO_SLEEP (use mdelay instead)

## Description

This routine places the adapter in diagnostic mode via the WriteSequence register and then performs a hard reset of adapter via the Diagnostic register. Adapter should be in ready state upon successful completion.

## Returns

1 hard reset successful 0 no reset performed because reset history bit set -2 enabling diagnostic mode failed  
-3 diagnostic reset failed

## Name

SendIocReset — Send IOCRreset request to MPT adapter.

## Synopsis

```
int SendIocReset (MPT_ADAPTER * ioc, u8 reset_type, int sleepFlag);
```

## Arguments

<i>ioc</i>	Pointer to MPT_ADAPTER structure
<i>reset_type</i>	reset type, expected values are MPI_FUNCTION_IOC_MESSAGE_UNIT_RESET or MPI_FUNCTION_IO_UNIT_RESET
<i>sleepFlag</i>	Specifies whether the process can sleep

## Description

Send IOCRreset request to the MPT adapter.

Returns 0 for success, non-zero for failure.

## Name

`initChainBuffers` — Allocate memory for and initialize chain buffers

## Synopsis

```
int initChainBuffers (MPT_ADAPTER * ioc);
```

## Arguments

*ioc* Pointer to MPT\_ADAPTER structure

## Description

Allocates memory for and initializes chain buffers, chain buffer control arrays and spinlock.

## Name

PrimeIocFifos — Initialize IOC request and reply FIFOs.

## Synopsis

```
int PrimeIocFifos (MPT_ADAPTER * ioc);
```

## Arguments

*ioc* Pointer to MPT\_ADAPTER structure

## Description

This routine allocates memory for the MPT reply and request frame pools (if necessary), and primes the IOC reply FIFO with reply frames.

Returns 0 for success, non-zero for failure.

## Name

`mpt_handshake_req_reply_wait` — Send MPT request to and receive reply from IOC via doorbell handshake method.

## Synopsis

```
int mpt_handshake_req_reply_wait (MPT_ADAPTER * ioc, int reqBytes, u32
* req, int replyBytes, u16 * u16reply, int maxwait, int sleepFlag);
```

## Arguments

<i>ioc</i>	Pointer to MPT_ADAPTER structure
<i>reqBytes</i>	Size of the request in bytes
<i>req</i>	Pointer to MPT request frame
<i>replyBytes</i>	Expected size of the reply in bytes
<i>u16reply</i>	Pointer to area where reply should be written
<i>maxwait</i>	Max wait time for a reply (in seconds)
<i>sleepFlag</i>	Specifies whether the process can sleep

## NOTES

It is the callers responsibility to byte-swap fields in the request which are greater than 1 byte in size. It is also the callers responsibility to byte-swap response fields which are greater than 1 byte in size.

Returns 0 for success, non-zero for failure.



## Name

WaitForDoorbellAck — Wait for IOC doorbell handshake acknowledge

## Synopsis

```
int WaitForDoorbellAck (MPT_ADAPTER * ioc, int howlong, int sleepFlag);
```

## Arguments

<i>ioc</i>	Pointer to MPT_ADAPTER structure
<i>howlong</i>	How long to wait (in seconds)
<i>sleepFlag</i>	Specifies whether the process can sleep

## Description

This routine waits (up to ~2 seconds max) for IOC doorbell handshake ACKnowledge, indicated by the IOP\_DOORBELL\_STATUS bit in its IntStatus register being clear.

Returns a negative value on failure, else wait loop count.

## Name

WaitForDoorbellInt — Wait for IOC to set its doorbell interrupt bit

## Synopsis

```
int WaitForDoorbellInt (MPT_ADAPTER * ioc, int howlong, int sleepFlag);
```

## Arguments

*ioc*            Pointer to MPT\_ADAPTER structure

*howlong*      How long to wait (in seconds)

*sleepFlag*    Specifies whether the process can sleep

## Description

This routine waits (up to ~2 seconds max) for IOC doorbell interrupt (MPI\_HIS\_DOORBELL\_INTERRUPT) to be set in the IntStatus register.

Returns a negative value on failure, else wait loop count.

## Name

WaitForDoorbellReply — Wait for and capture an IOC handshake reply.

## Synopsis

```
int WaitForDoorbellReply (MPT_ADAPTER * ioc, int howlong, int
sleepFlag);
```

## Arguments

<i>ioc</i>	Pointer to MPT_ADAPTER structure
<i>howlong</i>	How long to wait (in seconds)
<i>sleepFlag</i>	Specifies whether the process can sleep

## Description

This routine polls the IOC for a handshake reply, 16 bits at a time. Reply is cached to IOC private area large enough to hold a maximum of 128 bytes of reply data.

Returns a negative value on failure, else size of reply in WORDS.

## Name

GetLanConfigPages — Fetch LANConfig pages.

## Synopsis

```
int GetLanConfigPages (MPT_ADAPTER * ioc);
```

## Arguments

*ioc* Pointer to MPT\_ADAPTER structure

## Return

0 for success -ENOMEM if no memory available -EPERM if not allowed due to ISR context -EAGAIN if no msg frames currently available -EFAULT for non-successful reply or no reply (timeout)

## Name

GetIoUnitPage2 — Retrieve BIOS version and boot order information.

## Synopsis

```
int GetIoUnitPage2 (MPT_ADAPTER * ioc);
```

## Arguments

*ioc* Pointer to MPT\_ADAPTER structure

## Returns

0 for success -ENOMEM if no memory available -EPERM if not allowed due to ISR context -EAGAIN if no msg frames currently available -EFAULT for non-successful reply or no reply (timeout)

## Name

`mpt_GetScsiPortSettings` — read SCSI Port Page 0 and 2

## Synopsis

```
int mpt_GetScsiPortSettings (MPT_ADAPTER * ioc, int portnum);
```

## Arguments

*ioc*            Pointer to a Adapter Strucutre

*portnum*    IOC port number

## Return

-EFAULT if read of config page header fails or if no nvram If read of SCSI Port Page 0 fails, NVRAM = MPT\_HOST\_NVRAM\_INVALID (0xFFFFFFFF)

## Adapter settings

async, narrow Return 1 If read of SCSI Port Page 2 fails, Adapter settings valid NVRAM = MPT\_HOST\_NVRAM\_INVALID (0xFFFFFFFF) Return 1 Else Both valid Return 0 CHECK - what type of locking mechanisms should be used???

## Name

`mpt_readScsiDevicePageHeaders` — save version and length of SDP1

## Synopsis

```
int mpt_readScsiDevicePageHeaders (MPT_ADAPTER * ioc, int portnum);
```

## Arguments

*ioc*            Pointer to a Adapter Strucutre

*portnum*    IOC port number

## Return

-EFAULT if read of config page header fails or 0 if success.

## Name

`mpt_inactive_raid_list_free` — This clears this link list.

## Synopsis

```
void mpt_inactive_raid_list_free (MPT_ADAPTER * ioc);
```

## Arguments

*ioc* pointer to per adapter structure



## Name

`mpt_inactive_raid_volumes` — sets up link list of `phy_disk_nums` for devices belonging in an inactive volume

## Synopsis

```
void mpt_inactive_raid_volumes (MPT_ADAPTER * ioc, u8 channel, u8 id);
```

## Arguments

*ioc*            pointer to per adapter structure

*channel*       volume channel

*id*            volume target id

## Name

SendEventNotification — Send EventNotification (on or off) request to adapter

## Synopsis

```
int  SendEventNotification  (MPT_ADAPTER  *  ioc,  u8  EvSwitch,  int
sleepFlag);
```

## Arguments

<i>ioc</i>	Pointer to MPT_ADAPTER structure
<i>EvSwitch</i>	Event switch flags
<i>sleepFlag</i>	Specifies whether the process can sleep

## Name

SendEventAck — Send EventAck request to MPT adapter.

## Synopsis

```
int SendEventAck (MPT_ADAPTER * ioc, EventNotificationReply_t * evnp);
```

## Arguments

*ioc*    Pointer to MPT\_ADAPTER structure

*evnp*   Pointer to original EventNotification request

## Name

`mpt_ioc_reset` — Base cleanup for hard reset

## Synopsis

```
int mpt_ioc_reset (MPT_ADAPTER * ioc, int reset_phase);
```

## Arguments

*ioc*                      Pointer to the adapter structure

*reset\_phase*      Indicates pre- or post-reset functionality

## Remark

Frees resources with internally generated commands.

## Name

procmpt\_create — Create MPT\_PROCFS\_MPTBASEDIR entries.

## Synopsis

```
int procmpt_create ( void );
```

## Arguments

*void* no arguments

## Description

Returns 0 for success, non-zero for failure.

## Name

procmpt\_destroy — Tear down MPT\_PROCFS\_MPTBASEDIR entries.

## Synopsis

```
void procmpt_destroy ( void );
```

## Arguments

*void* no arguments

## Description

Returns 0 for success, non-zero for failure.

## Name

`mpt_SoftResetHandler` — Issues a less expensive reset

## Synopsis

```
int mpt_SoftResetHandler (MPT_ADAPTER * ioc, int sleepFlag);
```

## Arguments

*ioc*            Pointer to MPT\_ADAPTER structure

*sleepFlag*   Indicates if sleep or schedule must be called.

## Description

Returns 0 for SUCCESS or -1 if FAILED.

Message Unit Reset - instructs the IOC to reset the Reply Post and Free FIFO's. All the Message Frames on Reply Free FIFO are discarded. All posted buffers are freed, and event notification is turned off. IOC doesn't reply to any outstanding request. This will transfer IOC to READY state.

## Name

ProcessEventNotification — Route EventNotificationReply to all event handlers

## Synopsis

```
int      ProcessEventNotification      (MPT_ADAPTER      *      ioc,
EventNotificationReply_t * pEventReply, int * evHandlers);
```

## Arguments

*ioc*                Pointer to MPT\_ADAPTER structure

*pEventReply*    Pointer to EventNotification reply frame

*evHandlers*      Pointer to integer, number of event handlers

## Description

Routes a received EventNotificationReply to all currently registered event handlers. Returns sum of event handlers return values.



## Name

`mpt_fc_log_info` — Log information returned from Fibre Channel IOC.

## Synopsis

```
void mpt_fc_log_info (MPT_ADAPTER * ioc, u32 log_info);
```

## Arguments

*ioc*            Pointer to MPT\_ADAPTER structure

*log\_info*      U32 LogInfo reply word from the IOC

## Description

Refer to `lsi/mpi_log_fc.h`.

## Name

`mpt_spi_log_info` — Log information returned from SCSI Parallel IOC.

## Synopsis

```
void mpt_spi_log_info (MPT_ADAPTER * ioc, u32 log_info);
```

## Arguments

*ioc*            Pointer to MPT\_ADAPTER structure

*log\_info*    U32 LogInfo word from the IOC

## Description

Refer to `lsi/sp_log.h`.

## Name

`mpt_sas_log_info` — Log information returned from SAS IOC.

## Synopsis

```
void mpt_sas_log_info (MPT_ADAPTER * ioc, u32 log_info, u8 cb_idx);
```

## Arguments

<i>ioc</i>	Pointer to MPT_ADAPTER structure
<i>log_info</i>	U32 LogInfo reply word from the IOC
<i>cb_idx</i>	callback function's handle

## Description

Refer to `lsi/mpi_log_sas.h`.

## Name

`mpt_iocstatus_info_config` — IOCSTATUS information for config pages

## Synopsis

```
void mpt_iocstatus_info_config (MPT_ADAPTER * ioc, u32 ioc_status,
MPT_FRAME_HDR * mf);
```

## Arguments

<i>ioc</i>	Pointer to MPT_ADAPTER structure
<i>ioc_status</i>	U32 IOCStatus word from IOC
<i>mf</i>	Pointer to MPT request frame

## Description

Refer to `lsi/mpi.h`.

## Name

`mpt_iocstatus_info` — IOCSTATUS information returned from IOC.

## Synopsis

```
void mpt_iocstatus_info (MPT_ADAPTER * ioc, u32 ioc_status,
MPT_FRAME_HDR * mf);
```

## Arguments

<i>ioc</i>	Pointer to MPT_ADAPTER structure
<i>ioc_status</i>	U32 IOCStatus word from IOC
<i>mf</i>	Pointer to MPT request frame

## Description

Refer to `lsi/mpi.h`.

## Name

`fusion_init` — Fusion MPT base driver initialization routine.

## Synopsis

```
int fusion_init ( void );
```

## Arguments

*void* no arguments

## Description

Returns 0 for success, non-zero for failure.

## Name

`fusion_exit` — Perform driver unload cleanup.

## Synopsis

```
void __exit fusion_exit ( void );
```

## Arguments

*void* no arguments

## Description

This routine frees all resources associated with each MPT adapter and removes all `MPT_PROCFS_MPTBASEDIR` entries.

## Name

mptscsih\_info — Return information about MPT adapter

## Synopsis

```
const char * mptscsih_info (struct Scsi_Host * SHost);
```

## Arguments

*SHost*    Pointer to Scsi\_Host structure

## Description

(linux scsi\_host\_template.info routine)

Returns pointer to buffer where information was written.



## Name

mptscsih\_qcmd — Primary Fusion MPT SCSI initiator IO start routine.

## Synopsis

```
int mptscsih_qcmd (struct scsi_cmnd * SCpnt);
```

## Arguments

*SCpnt*    Pointer to scsi\_cmnd structure

## Description

(linux scsi\_host\_template.queuecommand routine) This is the primary SCSI IO start routine. Create a MPI SCSIIORequest from a linux scsi\_cmnd request and send it to the IOC.

Returns 0. (rtn value discarded by linux scsi mid-layer)

## Name

`mptscsih_IssueTaskMgmt` — Generic send Task Management function.

## Synopsis

```
int mptscsih_IssueTaskMgmt (MPT SCSI_HOST * hd, u8 type, u8 channel, u8  
id, u64 lun, int ctx2abort, ulong timeout);
```

## Arguments

<i>hd</i>	Pointer to MPT SCSI_HOST structure
<i>type</i>	Task Management type
<i>channel</i>	channel number for task management
<i>id</i>	Logical Target ID for reset (if appropriate)
<i>lun</i>	Logical Unit for reset (if appropriate)
<i>ctx2abort</i>	Context for the task to be aborted (if appropriate)
<i>timeout</i>	timeout for task management control

## Remark

`_HardResetHandler` can be invoked from an interrupt thread (timer) or a non-interrupt thread. In the former, must not call `schedule`.

Not all fields are meaningfull for all task types.

Returns 0 for SUCCESS, or FAILED.

## Name

mptscsih\_abort — Abort linux scsi\_cmnd routine, new\_eh variant

## Synopsis

```
int mptscsih_abort (struct scsi_cmnd * SCpnt);
```

## Arguments

*SCpnt* Pointer to scsi\_cmnd structure, IO to be aborted

## Description

(linux scsi\_host\_template.eh\_abort\_handler routine)

Returns SUCCESS or FAILED.

## Name

mptscsih\_dev\_reset — Perform a SCSI TARGET\_RESET! new\_eh variant

## Synopsis

```
int mptscsih_dev_reset (struct scsi_cmnd * SCpnt);
```

## Arguments

*SCpnt* Pointer to scsi\_cmnd structure, IO which reset is due to

## Description

(linux scsi\_host\_template.eh\_dev\_reset\_handler routine)

Returns SUCCESS or FAILED.

## Name

`mptscsih_bus_reset` — Perform a SCSI BUS\_RESET! new\_eh variant

## Synopsis

```
int mptscsih_bus_reset (struct scsi_cmnd * SCpnt);
```

## Arguments

*SCpnt* Pointer to `scsi_cmnd` structure, IO which reset is due to

## Description

(linux `scsi_host_template.eh_bus_reset_handler` routine)

Returns SUCCESS or FAILED.

## Name

mptscsih\_host\_reset — Perform a SCSI host adapter RESET (new\_eh variant)

## Synopsis

```
int mptscsih_host_reset (struct scsi_cmnd * SCpnt);
```

## Arguments

*SCpnt*    Pointer to scsi\_cmnd structure, IO which reset is due to

## Description

(linux scsi\_host\_template.eh\_host\_reset\_handler routine)

Returns SUCCESS or FAILED.

## Name

`mptscsih_taskmgmt_complete` — Registered with Fusion MPT base driver

## Synopsis

```
int mptscsih_taskmgmt_complete (MPT_ADAPTER * ioc, MPT_FRAME_HDR * mf,
MPT_FRAME_HDR * mr);
```

## Arguments

*ioc*    Pointer to MPT\_ADAPTER structure

*mf*     Pointer to SCSI task mgmt request frame

*mr*     Pointer to SCSI task mgmt reply frame

## Description

This routine is called from `mptbase.c::mpt_interrupt` at the completion of any SCSI task management request. This routine is registered with the MPT (base) driver at driver load/init time via the `mpt_register` API call.

Returns 1 indicating alloc'd request frame ptr should be freed.

## Name

`mptscsih_get_scsi_lookup` — retrieves scmd entry

## Synopsis

```
struct scsi_cmnd * mptscsih_get_scsi_lookup (MPT_ADAPTER * ioc, int i);
```

## Arguments

*ioc*    Pointer to MPT\_ADAPTER structure

*i*       index into the array

## Description

Returns the `scsi_cmd` pointer



## Name

mptscsih\_info\_scsiio — debug print info on reply frame

## Synopsis

```
void mptscsih_info_scsiio (MPT_ADAPTER * ioc, struct scsi_cmnd * sc,
SCSIIOReply_t * pScsiReply);
```

## Arguments

<i>ioc</i>	Pointer to MPT_ADAPTER structure
<i>sc</i>	original scsi cmnd pointer
<i>pScsiReply</i>	Pointer to MPT reply frame

## Description

MPT\_DEBUG\_REPLY needs to be enabled to obtain this info

Refer to lsi/mpi.h.

## Name

`mptscsih_getclear_scsi_lookup` — retrieves and clears `scmd` entry from `ScsiLookup[]` array list

## Synopsis

```
struct scsi_cmd * mptscsih_getclear_scsi_lookup (MPT_ADAPTER * ioc,
int i);
```

## Arguments

*ioc* Pointer to MPT\_ADAPTER structure

*i* index into the array

## Description

Returns the `scsi_cmd` pointer

## Name

mptscsih\_set\_scsi\_lookup — write a scmd entry into the ScsiLookup[] array list

## Synopsis

```
void mptscsih_set_scsi_lookup (MPT_ADAPTER * ioc, int i, struct
scsi_cmnd * scmd);
```

## Arguments

*ioc* Pointer to MPT\_ADAPTER structure

*i* index into the array

*scmd* scsi\_cmnd pointer

## Name

SCPNT\_TO\_LOOKUP\_IDX — searches for a given scmd in the ScsiLookup[] array list

## Synopsis

```
int SCPNT_TO_LOOKUP_IDX (MPT_ADAPTER * ioc, struct scsi_cmnd * sc);
```

## Arguments

*ioc* Pointer to MPT\_ADAPTER structure

*sc* scsi\_cmnd pointer

## Name

`mptscsih_get_completion_code` — get completion code from MPT request

## Synopsis

```
int mptscsih_get_completion_code (MPT_ADAPTER * ioc, MPT_FRAME_HDR *
req, MPT_FRAME_HDR * reply);
```

## Arguments

*ioc*     Pointer to MPT\_ADAPTER structure

*req*     Pointer to original MPT request frame

*reply*   Pointer to MPT reply frame (NULL if TurboReply)

## Name

`mptscsih_do_cmd` — Do internal command.

## Synopsis

```
int mptscsih_do_cmd (MPT SCSI_HOST * hd, INTERNAL_CMD * io);
```

## Arguments

*hd* MPT SCSI\_HOST pointer

*io* INTERNAL\_CMD pointer.

## Description

Issue the specified internally generated command and do command specific cleanup. For bus scan / DV only.

## NOTES

If command is Inquiry and status is good, initialize a target structure, save the data

## Remark

Single threaded access only.

## Return

< 0 if an illegal command or no resources

0 if good

> 0 if command complete but some type of completion error.

## Name

`mptscsih_synchronize_cache` — Send SYNCHRONIZE\_CACHE to all disks.

## Synopsis

```
void mptscsih_synchronize_cache (MPT SCSI_HOST * hd, VirtDevice *  
vdevice);
```

## Arguments

*hd*            Pointer to a SCSI HOST structure

*vdevice*    virtual target device

## Description

Uses the ISR, but with special processing. MUST be single-threaded.

## Name

`mptctl_syscall_down` — Down the MPT adapter syscall semaphore.

## Synopsis

```
int mptctl_syscall_down (MPT_ADAPTER * ioc, int nonblock);
```

## Arguments

*ioc*            Pointer to MPT adapter

*nonblock*    boolean, non-zero if O\_NONBLOCK is set

## Description

All of the `ioctl` commands can potentially sleep, which is illegal with a spinlock held, thus we perform mutual exclusion here.

Returns negative `errno` on error, or zero for success.



## Name

mptspi\_setTargetNegoParms — Update the target negotiation parameters

## Synopsis

```
void mptspi_setTargetNegoParms (MPT SCSI_HOST * hd, VirtTarget * target,  
struct scsi_device * sdev);
```

## Arguments

*hd*            Pointer to a SCSI Host Structure

*target*       per target private data

*sdev*          SCSI device

## Description

Update the target negotiation parameters based on the the Inquiry data, adapter capabilities, and NVRAM settings.

## Name

mptspi\_writeIOCPage4 — write IOC Page 4

## Synopsis

```
int mptspi_writeIOCPage4 (MPT_SCSI_HOST * hd, u8 channel, u8 id);
```

## Arguments

<i>hd</i>	Pointer to a SCSI Host Structure
<i>channel</i>	channel number
<i>id</i>	write IOC Page4 for this ID & Bus

## Return

-EAGAIN if unable to obtain a Message Frame or 0 if success.

## Remark

We do not wait for a return, write pages sequentially.

## Name

`mptspi_initTarget` — Target, LUN alloc/free functionality.

## Synopsis

```
void mptspi_initTarget (MPT SCSI_HOST * hd, VirtTarget * vtarget, struct
scsi_device * sdev);
```

## Arguments

*hd*            Pointer to MPT SCSI\_HOST structure

*vtarget*      per target private data

*sdev*         SCSI device

## NOTE

It's only SAFE to call this routine if data points to sane & valid STANDARD INQUIRY data!

Allocate and initialize memory for this target. Save inquiry data.

## Name

`mptspi_is_raid` — Determines whether target is belonging to volume

## Synopsis

```
int mptspi_is_raid (struct _MPT_SCSI_HOST * hd, u32 id);
```

## Arguments

*hd* Pointer to a SCSI HOST structure

*id* target device id

## Return

non-zero = true zero = false

## Name

mptspi\_print\_write\_nego — negotiation parameters debug info that is being sent

## Synopsis

```
void mptspi_print_write_nego (struct _MPT_SCSI_HOST * hd, struct  
scsi_target * starget, u32 ii);
```

## Arguments

*hd*            Pointer to a SCSI HOST structure

*starget*    SCSI target

*ii*           negotiation parameters

## Name

mptspi\_print\_read\_nego — negotiation parameters debug info that is being read

## Synopsis

```
void mptspi_print_read_nego (struct _MPT_SCSI_HOST * hd, struct  
scsi_target * starget, u32 ii);
```

## Arguments

*hd*            Pointer to a SCSI HOST structure

*starget*      SCSI target

*ii*           negotiation parameters

## Name

mptspi\_init — Register MPT adapter(s) as SCSI host(s) with SCSI mid-layer.

## Synopsis

```
int mptspi_init ( void );
```

## Arguments

*void* no arguments

## Description

Returns 0 for success, non-zero for failure.

## Name

mptspi\_exit — Unregisters MPT adapter(s)

## Synopsis

```
void __exit mptspi_exit ( void );
```

## Arguments

*void* no arguments



## Name

`mptfc_init` — Register MPT adapter(s) as SCSI host(s) with SCSI mid-layer.

## Synopsis

```
int mptfc_init ( void );
```

## Arguments

*void* no arguments

## Description

Returns 0 for success, non-zero for failure.

## Name

mptfc\_remove — Remove fc infrastructure for devices

## Synopsis

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

## Arguments

*pdev*    Pointer to pci\_dev structure

## Name

`mptfc_exit` — Unregisters MPT adapter(s)

## Synopsis

```
void __exit mptfc_exit ( void );
```

## Arguments

*void* no arguments

## Name

`lan_reply` — Handle all data sent from the hardware.

## Synopsis

```
int lan_reply (MPT_ADAPTER * ioc, MPT_FRAME_HDR * mf, MPT_FRAME_HDR *  
reply);
```

## Arguments

*ioc*      Pointer to MPT\_ADAPTER structure

*mf*       Pointer to original MPT request frame (NULL if TurboReply)

*reply*    Pointer to MPT reply frame

## Description

Returns 1 indicating original alloc'd request frame ptr should be freed, or 0 if it shouldn't.

---

# Chapter 5. Sound Devices

## Name

`snd_printk` — printk wrapper

## Synopsis

```
snd_printk ( fmt, args... );
```

## Arguments

*fmt*            format string

*args...*      variable arguments

## Description

Works like `printk` but prints the file and the line of the caller when configured with `CONFIG_SND_VERBOSE_PRINTK`.

## Name

`snd_printd` — debug printk

## Synopsis

```
snd_printd ( fmt, args... );
```

## Arguments

*fmt*            format string

*args...*      variable arguments

## Description

Works like `snd_printk` for debugging purposes. Ignored when `CONFIG_SND_DEBUG` is not set.

## Name

`snd_DEBUG` — give a BUG warning message and stack trace

## Synopsis

```
snd_DEBUG (void);
```

## Arguments

None

## Description

Calls `WARN` if `CONFIG_SND_DEBUG` is set. Ignored when `CONFIG_SND_DEBUG` is not set.



## Name

`snd_printd_ratelimit` —

## Synopsis

```
snd_printd_ratelimit (void);
```

## Arguments

None

## Name

`snd_DEBUG_ON` — debugging check macro

## Synopsis

```
snd_DEBUG_ON ( cond );
```

## Arguments

*cond*    condition to evaluate

## Description

Has the same behavior as `WARN_ON` when `CONFIG_SND_DEBUG` is set, otherwise just evaluates the conditional and returns the value.

## Name

`snd_printdd` — debug printk

## Synopsis

```
snd_printdd ( format, args... );
```

## Arguments

*format*     format string

*args...*   variable arguments

## Description

Works like `snd_printk` for debugging purposes. Ignored when `CONFIG_SND_DEBUG_VERBOSE` is not set.

## Name

`register_sound_special_device` — register a special sound node

## Synopsis

```
int register_sound_special_device (const struct file_operations * fops,
int unit, struct device * dev);
```

## Arguments

*fops*    File operations for the driver

*unit*    Unit number to allocate

*dev*    device pointer

## Description

Allocate a special sound device by minor number from the sound subsystem.

## Return

The allocated number is returned on success. On failure, a negative error code is returned.

## Name

`register_sound_mixer` — register a mixer device

## Synopsis

```
int register_sound_mixer (const struct file_operations * fops, int dev);
```

## Arguments

*fops*    File operations for the driver

*dev*    Unit number to allocate

## Description

Allocate a mixer device. Unit is the number of the mixer requested. Pass -1 to request the next free mixer unit.

## Return

On success, the allocated number is returned. On failure, a negative error code is returned.

## Name

`register_sound_midi` — register a midi device

## Synopsis

```
int register_sound_midi (const struct file_operations * fops, int dev);
```

## Arguments

*fops*    File operations for the driver

*dev*     Unit number to allocate

## Description

Allocate a midi device. Unit is the number of the midi device requested. Pass -1 to request the next free midi unit.

## Return

On success, the allocated number is returned. On failure, a negative error code is returned.

## Name

`register_sound_dsp` — register a DSP device

## Synopsis

```
int register_sound_dsp (const struct file_operations * fops, int dev);
```

## Arguments

*fops*    File operations for the driver

*dev*     Unit number to allocate

## Description

Allocate a DSP device. Unit is the number of the DSP requested. Pass -1 to request the next free DSP unit.

This function allocates both the audio and dsp device entries together and will always allocate them as a matching pair - eg dsp3/audio3

## Return

On success, the allocated number is returned. On failure, a negative error code is returned.

## Name

`unregister_sound_special` — unregister a special sound device

## Synopsis

```
void unregister_sound_special (int unit);
```

## Arguments

*unit*    unit number to allocate

## Description

Release a sound device that was allocated with `register_sound_special`. The unit passed is the return value from the register function.



## Name

`unregister_sound_mixer` — unregister a mixer

## Synopsis

```
void unregister_sound_mixer (int unit);
```

## Arguments

*unit*    unit number to allocate

## Description

Release a sound device that was allocated with `register_sound_mixer`. The unit passed is the return value from the register function.

## Name

`unregister_sound_midi` — unregister a midi device

## Synopsis

```
void unregister_sound_midi (int unit);
```

## Arguments

*unit*    unit number to allocate

## Description

Release a sound device that was allocated with `register_sound_midi`. The unit passed is the return value from the register function.

## Name

`unregister_sound_dsp` — unregister a DSP device

## Synopsis

```
void unregister_sound_dsp (int unit);
```

## Arguments

*unit*    unit number to allocate

## Description

Release a sound device that was allocated with `register_sound_dsp`. The unit passed is the return value from the register function.

Both of the allocated units are released together automatically.

## Name

`snd_pcm_stream_linked` — Check whether the substream is linked with others

## Synopsis

```
int snd_pcm_stream_linked (struct snd_pcm_substream * substream);
```

## Arguments

*substream*   substream to check

## Description

Returns true if the given substream is being linked with others.

## Name

`snd_pcm_stream_lock_irqsave` — Lock the PCM stream

## Synopsis

```
snd_pcm_stream_lock_irqsave ( substream, flags );
```

## Arguments

*substream*    PCM substream

*flags*        irq flags

## Description

This locks the PCM stream like `snd_pcm_stream_lock` but with the local IRQ (only when `nonatomic` is false). In nonatomic case, this is identical as `snd_pcm_stream_lock`.

## Name

`snd_pcm_group_for_each_entry` — iterate over the linked substreams

## Synopsis

```
snd_pcm_group_for_each_entry ( s, substream );
```

## Arguments

*s*                    the iterator

*substream*    the substream

## Description

Iterate over the all linked substreams to the given *substream*. When *substream* isn't linked with any others, this gives returns *substream* itself once.

## Name

`snd_pcm_running` — Check whether the substream is in a running state

## Synopsis

```
int snd_pcm_running (struct snd_pcm_substream * substream);
```

## Arguments

*substream*    substream to check

## Description

Returns true if the given substream is in the state `RUNNING`, or in the state `DRAINING` for playback.

## Name

`bytes_to_samples` — Unit conversion of the size from bytes to samples

## Synopsis

```
ssize_t bytes_to_samples (struct snd_pcm_runtime * runtime, ssize_t  
size);
```

## Arguments

*runtime*    PCM runtime instance

*size*       size in bytes



## Name

`bytes_to_frames` — Unit conversion of the size from bytes to frames

## Synopsis

```
snd_pcm_sframes_t bytes_to_frames (struct snd_pcm_runtime * runtime,  
ssize_t size);
```

## Arguments

*runtime*    PCM runtime instance

*size*       size in bytes

## Name

`samples_to_bytes` — Unit conversion of the size from samples to bytes

## Synopsis

```
ssize_t samples_to_bytes (struct snd_pcm_runtime * runtime, ssize_t  
size);
```

## Arguments

*runtime*    PCM runtime instance

*size*       size in samples

## Name

`frames_to_bytes` — Unit conversion of the size from frames to bytes

## Synopsis

```
ssize_t frames_to_bytes (struct snd_pcm_runtime * runtime,  
snd_pcm_sframes_t size);
```

## Arguments

*runtime*    PCM runtime instance

*size*       size in frames

## Name

`frame_aligned` — Check whether the byte size is aligned to frames

## Synopsis

```
int frame_aligned (struct snd_pcm_runtime * runtime, ssize_t bytes);
```

## Arguments

*runtime*    PCM runtime instance

*bytes*      size in bytes

## Name

`snd_pcm_lib_buffer_bytes` — Get the buffer size of the current PCM in bytes

## Synopsis

```
size_t snd_pcm_lib_buffer_bytes (struct snd_pcm_substream * substream);
```

## Arguments

*substream*   PCM substream

## Name

`snd_pcm_lib_period_bytes` — Get the period size of the current PCM in bytes

## Synopsis

```
size_t snd_pcm_lib_period_bytes (struct snd_pcm_substream * substream);
```

## Arguments

*substream*   PCM substream

## Name

`snd_pcm_playback_avail` — Get the available (writable) space for playback

## Synopsis

```
snd_pcm_uframes_t snd_pcm_playback_avail (struct snd_pcm_runtime *  
runtime);
```

## Arguments

*runtime* PCM runtime instance

## Description

Result is between 0 ... (boundary - 1)

## Name

`snd_pcm_capture_avail` — Get the available (readable) space for capture

## Synopsis

```
snd_pcm_uframes_t  snd_pcm_capture_avail (struct  snd_pcm_runtime  *  
runtime);
```

## Arguments

*runtime* PCM runtime instance

## Description

Result is between 0 ... (boundary - 1)



## Name

`snd_pcm_playback_hw_avail` — Get the queued space for playback

## Synopsis

```
snd_pcm_sframes_t snd_pcm_playback_hw_avail (struct snd_pcm_runtime *  
runtime);
```

## Arguments

*runtime*    PCM runtime instance

## Name

`snd_pcm_capture_hw_avail` — Get the free space for capture

## Synopsis

```
snd_pcm_sframes_t snd_pcm_capture_hw_avail (struct snd_pcm_runtime *  
runtime);
```

## Arguments

*runtime*    PCM runtime instance

## Name

`snd_pcm_playback_ready` — check whether the playback buffer is available

## Synopsis

```
int snd_pcm_playback_ready (struct snd_pcm_substream * substream);
```

## Arguments

*substream* the pcm substream instance

## Description

Checks whether enough free space is available on the playback buffer.

## Return

Non-zero if available, or zero if not.

## Name

`snd_pcm_capture_ready` — check whether the capture buffer is available

## Synopsis

```
int snd_pcm_capture_ready (struct snd_pcm_substream * substream);
```

## Arguments

*substream*    the pcm substream instance

## Description

Checks whether enough capture data is available on the capture buffer.

## Return

Non-zero if available, or zero if not.

## Name

`snd_pcm_playback_data` — check whether any data exists on the playback buffer

## Synopsis

```
int snd_pcm_playback_data (struct snd_pcm_substream * substream);
```

## Arguments

*substream* the pcm substream instance

## Description

Checks whether any data exists on the playback buffer.

## Return

Non-zero if any data exists, or zero if not. If `stop_threshold` is bigger or equal to `boundary`, then this function returns always non-zero.

## Name

`snd_pcm_playback_empty` — check whether the playback buffer is empty

## Synopsis

```
int snd_pcm_playback_empty (struct snd_pcm_substream * substream);
```

## Arguments

*substream* the pcm substream instance

## Description

Checks whether the playback buffer is empty.

## Return

Non-zero if empty, or zero if not.

## Name

`snd_pcm_capture_empty` — check whether the capture buffer is empty

## Synopsis

```
int snd_pcm_capture_empty (struct snd_pcm_substream * substream);
```

## Arguments

*substream* the pcm substream instance

## Description

Checks whether the capture buffer is empty.

## Return

Non-zero if empty, or zero if not.

## Name

`snd_pcm_trigger_done` — Mark the master substream

## Synopsis

```
void snd_pcm_trigger_done (struct snd_pcm_substream * substream, struct  
snd_pcm_substream * master);
```

## Arguments

*substream*    the pcm substream instance

*master*        the linked master substream

## Description

When multiple substreams of the same card are linked and the hardware supports the single-shot operation, the driver calls this in the loop in `snd_pcm_group_for_each_entry` for marking the substream as “done”. Then most of trigger operations are performed only to the given master substream.

The `trigger_master` mark is cleared at timestamp updates at the end of trigger operations.



## Name

`params_channels` — Get the number of channels from the hw params

## Synopsis

```
unsigned int params_channels (const struct snd_pcm_hw_params * p);
```

## Arguments

*p* hw params

## Name

`params_rate` — Get the sample rate from the hw params

## Synopsis

```
unsigned int params_rate (const struct snd_pcm_hw_params * p);
```

## Arguments

*p* hw params

## Name

`params_period_size` — Get the period size (in frames) from the hw params

## Synopsis

```
unsigned int params_period_size (const struct snd_pcm_hw_params * p);
```

## Arguments

*p* hw params

## Name

params\_periods — Get the number of periods from the hw params

## Synopsis

```
unsigned int params_periods (const struct snd_pcm_hw_params * p);
```

## Arguments

*p* hw params

## Name

`params_buffer_size` — Get the buffer size (in frames) from the hw params

## Synopsis

```
unsigned int params_buffer_size (const struct snd_pcm_hw_params * p);
```

## Arguments

*p* hw params

## Name

`params_buffer_bytes` — Get the buffer size (in bytes) from the hw params

## Synopsis

```
unsigned int params_buffer_bytes (const struct snd_pcm_hw_params * p);
```

## Arguments

*p* hw params

## Name

`snd_pcm_hw_constraint_single` — Constrain parameter to a single value

## Synopsis

```
int snd_pcm_hw_constraint_single (struct snd_pcm_runtime * runtime,
snd_pcm_hw_param_t var, unsigned int val);
```

## Arguments

<i>runtime</i>	PCM runtime instance
<i>var</i>	The <code>hw_params</code> variable to constrain
<i>val</i>	The value to constrain to

## Return

Positive if the value is changed, zero if it's not changed, or a negative error code.

## Name

`snd_pcm_format_cpu_endian` — Check the PCM format is CPU-endian

## Synopsis

```
int snd_pcm_format_cpu_endian (snd_pcm_format_t format);
```

## Arguments

*format* the format to check

## Return

1 if the given PCM format is CPU-endian, 0 if opposite, or a negative error code if endian not specified.



## Name

`snd_pcm_set_runtime_buffer` — Set the PCM runtime buffer

## Synopsis

```
void snd_pcm_set_runtime_buffer (struct snd_pcm_substream * substream,  
struct snd_dma_buffer * bufp);
```

## Arguments

*substream*    PCM substream to set

*bufp*        the buffer information, NULL to clear

## Description

Copy the buffer information to `runtime->dma_buffer` when *bufp* is non-NULL. Otherwise it clears the current buffer information.

## Name

`snd_pcm_gettime` — Fill the timespec depending on the timestamp mode

## Synopsis

```
void snd_pcm_gettime (struct snd_pcm_runtime * runtime, struct timespec  
* tv);
```

## Arguments

*runtime*    PCM runtime instance

*tv*        timespec to fill

## Name

`snd_pcm_lib_alloc_vmalloc_buffer` — allocate virtual DMA buffer

## Synopsis

```
int  snd_pcm_lib_alloc_vmalloc_buffer (struct  snd_pcm_substream  *  
    substream, size_t size);
```

## Arguments

*substream*    the substream to allocate the buffer to

*size*            the requested buffer size, in bytes

## Description

Allocates the PCM substream buffer using `vmalloc`, i.e., the memory is contiguous in kernel virtual space, but not in physical memory. Use this if the buffer is accessed by kernel code but not by device DMA.

## Return

1 if the buffer was changed, 0 if not changed, or a negative error code.

## Name

`snd_pcm_lib_alloc_vmalloc_32_buffer` — allocate 32-bit-addressable buffer

## Synopsis

```
int snd_pcm_lib_alloc_vmalloc_32_buffer (struct snd_pcm_substream *  
    substream, size_t size);
```

## Arguments

*substream*    the substream to allocate the buffer to

*size*            the requested buffer size, in bytes

## Description

This function works like `snd_pcm_lib_alloc_vmalloc_buffer`, but uses `vmalloc_32`, i.e., the pages are allocated from 32-bit-addressable memory.

## Return

1 if the buffer was changed, 0 if not changed, or a negative error code.

## Name

`snd_pcm_sgbuf_get_addr` — Get the DMA address at the corresponding offset

## Synopsis

```
dma_addr_t    snd_pcm_sgbuf_get_addr    (struct    snd_pcm_substream    *  
substream, unsigned int ofs);
```

## Arguments

*substream* PCM substream

*ofs* byte offset

## Name

`snd_pcm_sgbuf_get_ptr` — Get the virtual address at the corresponding offset

## Synopsis

```
void * snd_pcm_sgbuf_get_ptr (struct snd_pcm_substream * substream,  
unsigned int ofs);
```

## Arguments

*substream*    PCM substream

*ofs*            byte offset

## Name

`snd_pcm_sgbuf_get_chunk_size` — Compute the max size that fits within the contig. page from the given size

## Synopsis

```
unsigned int snd_pcm_sgbuf_get_chunk_size (struct snd_pcm_substream *  
substream, unsigned int ofs, unsigned int size);
```

## Arguments

<i>substream</i>	PCM substream
<i>ofs</i>	byte offset
<i>size</i>	byte size to examine

## Name

`snd_pcm_mmap_data_open` — increase the mmap counter

## Synopsis

```
void snd_pcm_mmap_data_open (struct vm_area_struct * area);
```

## Arguments

*area* VMA

## Description

PCM mmap callback should handle this counter properly



## Name

`snd_pcm_mmap_data_close` — decrease the mmap counter

## Synopsis

```
void snd_pcm_mmap_data_close (struct vm_area_struct * area);
```

## Arguments

*area* VMA

## Description

PCM mmap callback should handle this counter properly

## Name

`snd_pcm_limit_isa_dma_size` — Get the max size fitting with ISA DMA transfer

## Synopsis

```
void snd_pcm_limit_isa_dma_size (int dma, size_t * max);
```

## Arguments

*dma*   DMA number

*max*   pointer to store the max size

## Name

`snd_pcm_stream_str` — Get a string naming the direction of a stream

## Synopsis

```
const char * snd_pcm_stream_str (struct snd_pcm_substream * substream);
```

## Arguments

*substream* the pcm substream instance

## Return

A string naming the direction of the stream.

## Name

`snd_pcm_chmap_substream` — get the PCM substream assigned to the given chmap info

## Synopsis

```
struct    snd_pcm_substream    *    snd_pcm_chmap_substream    (struct  
snd_pcm_chmap * info, unsigned int idx);
```

## Arguments

*info* chmap information

*idx* the substream number index

## Name

`pcm_format_to_bits` — Strong-typed conversion of `pcm_format` to bitwise

## Synopsis

```
u64 pcm_format_to_bits (snd_pcm_format_t pcm_format);
```

## Arguments

*pcm\_format*    PCM format

## Name

`snd_pcm_format_name` — Return a name string for the given PCM format

## Synopsis

```
const char * snd_pcm_format_name (snd_pcm_format_t format);
```

## Arguments

*format*    PCM format

## Name

`snd_pcm_new_stream` — create a new PCM stream

## Synopsis

```
int snd_pcm_new_stream (struct snd_pcm * pcm, int stream, int
substream_count);
```

## Arguments

<i>pcm</i>	the pcm instance
<i>stream</i>	the stream direction, <code>SNDRV_PCM_STREAM_XXX</code>
<i>substream_count</i>	the number of substreams

## Description

Creates a new stream for the pcm. The corresponding stream on the pcm must have been empty before calling this, i.e. zero must be given to the argument of `snd_pcm_new`.

## Return

Zero if successful, or a negative error code on failure.

## Name

`snd_pcm_new` — create a new PCM instance

## Synopsis

```
int snd_pcm_new (struct snd_card * card, const char * id, int device,  
int playback_count, int capture_count, struct snd_pcm ** rpcm);
```

## Arguments

<i>card</i>	the card instance
<i>id</i>	the id string
<i>device</i>	the device index (zero based)
<i>playback_count</i>	the number of substreams for playback
<i>capture_count</i>	the number of substreams for capture
<i>rpcm</i>	the pointer to store the new pcm instance

## Description

Creates a new PCM instance.

The pcm operators have to be set afterwards to the new instance via `snd_pcm_set_ops`.

## Return

Zero if successful, or a negative error code on failure.



## Name

`snd_pcm_new_internal` — create a new internal PCM instance

## Synopsis

```
int snd_pcm_new_internal (struct snd_card * card, const char * id, int
device, int playback_count, int capture_count, struct snd_pcm ** rpcm);
```

## Arguments

<i>card</i>	the card instance
<i>id</i>	the id string
<i>device</i>	the device index (zero based - shared with normal PCM's)
<i>playback_count</i>	the number of substreams for playback
<i>capture_count</i>	the number of substreams for capture
<i>rpcm</i>	the pointer to store the new pcm instance

## Description

Creates a new internal PCM instance with no userspace device or procfs entries. This is used by ASoC Back End PCM's in order to create a PCM that will only be used internally by kernel drivers. i.e. it cannot be opened by userspace. It provides existing ASoC components drivers with a substream and access to any private data.

The pcm operators have to be set afterwards to the new instance via `snd_pcm_set_ops`.

## Return

Zero if successful, or a negative error code on failure.

## Name

`snd_pcm_notify` — Add/remove the notify list

## Synopsis

```
int snd_pcm_notify (struct snd_pcm_notify * notify, int nfree);
```

## Arguments

*notify*    PCM notify list

*nfree*     0 = register, 1 = unregister

## Description

This adds the given notifier to the global list so that the callback is called for each registered PCM devices. This exists only for PCM OSS emulation, so far.

## Name

`snd_device_new` — create an ALSA device component

## Synopsis

```
int snd_device_new (struct snd_card * card, enum snd_device_type type,  
void * device_data, struct snd_device_ops * ops);
```

## Arguments

<i>card</i>	the card instance
<i>type</i>	the device type, <code>SNDRV_DEV_XXX</code>
<i>device_data</i>	the data pointer of this device
<i>ops</i>	the operator table

## Description

Creates a new device component for the given data pointer. The device will be assigned to the card and managed together by the card.

The data pointer plays a role as the identifier, too, so the pointer address must be unique and unchanged.

## Return

Zero if successful, or a negative error code on failure.

## Name

`snd_device_disconnect` — disconnect the device

## Synopsis

```
void snd_device_disconnect (struct snd_card * card, void * device_data);
```

## Arguments

*card*                    the card instance

*device\_data*    the data pointer to disconnect

## Description

Turns the device into the disconnection state, invoking `dev_disconnect` callback, if the device was already registered.

Usually called from `snd_card_disconnect`.

## Return

Zero if successful, or a negative error code on failure or if the device not found.

## Name

`snd_device_free` — release the device from the card

## Synopsis

```
void snd_device_free (struct snd_card * card, void * device_data);
```

## Arguments

*card*                    the card instance

*device\_data*    the data pointer to release

## Description

Removes the device from the list on the card and invokes the callbacks, `dev_disconnect` and `dev_free`, corresponding to the state. Then release the device.

## Name

`snd_device_register` — register the device

## Synopsis

```
int snd_device_register (struct snd_card * card, void * device_data);
```

## Arguments

*card*                    the card instance

*device\_data*    the data pointer to register

## Description

Registers the device which was already created via `snd_device_new`. Usually this is called from `snd_card_register`, but it can be called later if any new devices are created after invocation of `snd_card_register`.

## Return

Zero if successful, or a negative error code on failure or if the device not found.

## Name

`snd_info_get_line` — read one line from the procfs buffer

## Synopsis

```
int snd_info_get_line (struct snd_info_buffer * buffer, char * line,  
int len);
```

## Arguments

*buffer*    the procfs buffer

*line*      the buffer to store

*len*        the max. buffer size

## Description

Reads one line from the buffer and stores the string.

## Return

Zero if successful, or 1 if error or EOF.

## Name

`snd_info_get_str` — parse a string token

## Synopsis

```
const char * snd_info_get_str (char * dest, const char * src, int len);
```

## Arguments

*dest*    the buffer to store the string token

*src*     the original string

*len*     the max. length of token - 1

## Description

Parses the original string and copy a token to the given string buffer.

## Return

The updated pointer of the original string so that it can be used for the next call.



## Name

`snd_info_create_module_entry` — create an info entry for the given module

## Synopsis

```
struct snd_info_entry * snd_info_create_module_entry (struct module *  
module, const char * name, struct snd_info_entry * parent);
```

## Arguments

*module*    the module pointer

*name*      the file name

*parent*    the parent directory

## Description

Creates a new info entry and assigns it to the given module.

## Return

The pointer of the new instance, or NULL on failure.

## Name

`snd_info_create_card_entry` — create an info entry for the given card

## Synopsis

```
struct snd_info_entry * snd_info_create_card_entry (struct snd_card *  
card, const char * name, struct snd_info_entry * parent);
```

## Arguments

*card*      the card instance

*name*      the file name

*parent*    the parent directory

## Description

Creates a new info entry and assigns it to the given card.

## Return

The pointer of the new instance, or NULL on failure.

## Name

`snd_info_free_entry` — release the info entry

## Synopsis

```
void snd_info_free_entry (struct snd_info_entry * entry);
```

## Arguments

*entry* the info entry

## Description

Releases the info entry.

## Name

`snd_info_register` — register the info entry

## Synopsis

```
int snd_info_register (struct snd_info_entry * entry);
```

## Arguments

*entry* the info entry

## Description

Registers the proc info entry.

## Return

Zero if successful, or a negative error code on failure.

## Name

`snd_rawmidi_receive` — receive the input data from the device

## Synopsis

```
int snd_rawmidi_receive (struct snd_rawmidi_substream * substream, const
unsigned char * buffer, int count);
```

## Arguments

*substream*    the rawmidi substream

*buffer*       the buffer pointer

*count*        the data size to read

## Description

Reads the data from the internal buffer.

## Return

The size of read data, or a negative error code on failure.

## Name

`snd_rawmidi_transmit_empty` — check whether the output buffer is empty

## Synopsis

```
int  snd_rawmidi_transmit_empty (struct  snd_rawmidi_substream  *  
    substream);
```

## Arguments

*substream* the rawmidi substream

## Return

1 if the internal output buffer is empty, 0 if not.

## Name

`__snd_rawmidi_transmit_peek` — copy data from the internal buffer

## Synopsis

```
int  __snd_rawmidi_transmit_peek (struct  snd_rawmidi_substream  *  
    substream, unsigned char * buffer, int count);
```

## Arguments

*substream*    the rawmidi substream

*buffer*       the buffer pointer

*count*        data size to transfer

## Description

This is a variant of `snd_rawmidi_transmit_peek` without spinlock.

## Name

`snd_rawmidi_transmit_peek` — copy data from the internal buffer

## Synopsis

```
int  snd_rawmidi_transmit_peek (struct  snd_rawmidi_substream  *  
    substream, unsigned char * buffer, int count);
```

## Arguments

*substream* the rawmidi substream

*buffer* the buffer pointer

*count* data size to transfer

## Description

Copies data from the internal output buffer to the given buffer.

Call this in the interrupt handler when the midi output is ready, and call `snd_rawmidi_transmit_ack` after the transmission is finished.

## Return

The size of copied data, or a negative error code on failure.



## Name

`__snd_rawmidi_transmit_ack` — acknowledge the transmission

## Synopsis

```
int  __snd_rawmidi_transmit_ack (struct snd_rawmidi_substream *  
    substream, int count);
```

## Arguments

*substream*    the rawmidi substream

*count*        the transferred count

## Description

This is a variant of `__snd_rawmidi_transmit_ack` without spinlock.

## Name

`snd_rawmidi_transmit_ack` — acknowledge the transmission

## Synopsis

```
int snd_rawmidi_transmit_ack (struct snd_rawmidi_substream * substream,
int count);
```

## Arguments

*substream*    the rawmidi substream

*count*        the transferred count

## Description

Advances the hardware pointer for the internal output buffer with the given size and updates the condition. Call after the transmission is finished.

## Return

The advanced size if successful, or a negative error code on failure.

## Name

`snd_rawmidi_transmit` — copy from the buffer to the device

## Synopsis

```
int snd_rawmidi_transmit (struct snd_rawmidi_substream * substream,  
unsigned char * buffer, int count);
```

## Arguments

*substream*    the rawmidi substream

*buffer*       the buffer pointer

*count*        the data size to transfer

## Description

Copies data from the buffer to the device and advances the pointer.

## Return

The copied size if successful, or a negative error code on failure.

## Name

`snd_rawmidi_new` — create a rawmidi instance

## Synopsis

```
int snd_rawmidi_new (struct snd_card * card, char * id, int device, int
output_count, int input_count, struct snd_rawmidi ** rrawmidi);
```

## Arguments

<i>card</i>	the card instance
<i>id</i>	the id string
<i>device</i>	the device index
<i>output_count</i>	the number of output streams
<i>input_count</i>	the number of input streams
<i>rrawmidi</i>	the pointer to store the new rawmidi instance

## Description

Creates a new rawmidi instance. Use `snd_rawmidi_set_ops` to set the operators to the new instance.

## Return

Zero if successful, or a negative error code on failure.

## Name

`snd_rawmidi_set_ops` — set the rawmidi operators

## Synopsis

```
void snd_rawmidi_set_ops (struct snd_rawmidi * rmidi, int stream, struct  
snd_rawmidi_ops * ops);
```

## Arguments

*rmidi*     the rawmidi instance

*stream*   the stream direction, `SNDRV_RAWMIDI_STREAM_XXX`

*ops*       the operator table

## Description

Sets the rawmidi operators for the given stream direction.

## Name

`snd_request_card` — try to load the card module

## Synopsis

```
void snd_request_card (int card);
```

## Arguments

*card* the card number

## Description

Tries to load the module “snd-card-X” for the given card number via `request_module`. Returns immediately if already loaded.

## Name

`snd_lookup_minor_data` — get user data of a registered device

## Synopsis

```
void * snd_lookup_minor_data (unsigned int minor, int type);
```

## Arguments

*minor*    the minor number

*type*    device type (SNDRV\_DEVICE\_TYPE\_XXX)

## Description

Checks that a minor device with the specified type is registered, and returns its user data pointer.

This function increments the reference counter of the card instance if an associated instance with the given minor number and type is found. The caller must call `snd_card_unref` appropriately later.

## Return

The user data pointer if the specified device is found. NULL otherwise.

## Name

`snd_register_device` — Register the ALSA device file for the card

## Synopsis

```
int snd_register_device (int type, struct snd_card * card, int dev,  
const struct file_operations * f_ops, void * private_data, struct device  
* device);
```

## Arguments

<i>type</i>	the device type, SNDRV_DEVICE_TYPE_XXX
<i>card</i>	the card instance
<i>dev</i>	the device index
<i>f_ops</i>	the file operations
<i>private_data</i>	user pointer for <i>f_ops</i> ->open
<i>device</i>	the device to register

## Description

Registers an ALSA device file for the given card. The operators have to be set in *reg* parameter.

## Return

Zero if successful, or a negative error code on failure.



## Name

`snd_unregister_device` — unregister the device on the given card

## Synopsis

```
int snd_unregister_device (struct device * dev);
```

## Arguments

*dev* the device instance

## Description

Unregisters the device file already registered via `snd_register_device`.

## Return

Zero if successful, or a negative error code on failure.

## Name

`copy_to_user_fromio` — copy data from mmio-space to user-space

## Synopsis

```
int copy_to_user_fromio (void __user * dst, const volatile void __iomem
* src, size_t count);
```

## Arguments

*dst*      the destination pointer on user-space

*src*      the source pointer on mmio

*count*    the data size to copy in bytes

## Description

Copies the data from mmio-space to user-space.

## Return

Zero if successful, or non-zero on failure.

## Name

`copy_from_user_toio` — copy data from user-space to mmio-space

## Synopsis

```
int copy_from_user_toio (volatile void __iomem * dst, const void __user
* src, size_t count);
```

## Arguments

*dst*      the destination pointer on mmio-space

*src*      the source pointer on user-space

*count*    the data size to copy in bytes

## Description

Copies the data from user-space to mmio-space.

## Return

Zero if successful, or non-zero on failure.

## Name

`snd_pcm_lib_preallocate_free_for_all` — release all pre-allocated buffers on the pcm

## Synopsis

```
int snd_pcm_lib_preallocate_free_for_all (struct snd_pcm * pcm);
```

## Arguments

*pcm* the pcm instance

## Description

Releases all the pre-allocated buffers on the given pcm.

## Return

Zero if successful, or a negative error code on failure.

## Name

`snd_pcm_lib_preallocate_pages` — pre-allocation for the given DMA type

## Synopsis

```
int    snd_pcm_lib_preallocate_pages    (struct    snd_pcm_substream    *  
substream, int type, struct device * data, size_t size, size_t max);
```

## Arguments

<i>substream</i>	the pcm substream instance
<i>type</i>	DMA type (SNDRV_DMA_TYPE_*)
<i>data</i>	DMA type dependent data
<i>size</i>	the requested pre-allocation size in bytes
<i>max</i>	the max. allowed pre-allocation size

## Description

Do pre-allocation for the given DMA buffer type.

## Return

Zero if successful, or a negative error code on failure.

## Name

`snd_pcm_lib_preallocate_pages_for_all` — pre-allocation for continuous memory type (all substreams)

## Synopsis

```
int snd_pcm_lib_preallocate_pages_for_all (struct snd_pcm * pcm, int
type, void * data, size_t size, size_t max);
```

## Arguments

*pcm*     the pcm instance

*type*    DMA type (SNDRV\_DMA\_TYPE\_\*)

*data*    DMA type dependent data

*size*    the requested pre-allocation size in bytes

*max*     the max. allowed pre-allocation size

## Description

Do pre-allocation to all substreams of the given pcm for the specified DMA type.

## Return

Zero if successful, or a negative error code on failure.

## Name

`snd_pcm_sgbuf_ops_page` — get the page struct at the given offset

## Synopsis

```
struct page * snd_pcm_sgbuf_ops_page (struct snd_pcm_substream *  
substream, unsigned long offset);
```

## Arguments

*substream*    the pcm substream instance

*offset*       the buffer offset

## Description

Used as the page callback of PCM ops.

## Return

The page struct at the given buffer offset. NULL on failure.

## Name

`snd_pcm_lib_malloc_pages` — allocate the DMA buffer

## Synopsis

```
int snd_pcm_lib_malloc_pages (struct snd_pcm_substream * substream,
size_t size);
```

## Arguments

*substream* the substream to allocate the DMA buffer to

*size* the requested buffer size in bytes

## Description

Allocates the DMA buffer on the BUS type given earlier to `snd_pcm_lib_preallocate_XXX_pages`.

## Return

1 if the buffer is changed, 0 if not changed, or a negative code on failure.



## Name

`snd_pcm_lib_free_pages` — release the allocated DMA buffer.

## Synopsis

```
int snd_pcm_lib_free_pages (struct snd_pcm_substream * substream);
```

## Arguments

*substream* the substream to release the DMA buffer

## Description

Releases the DMA buffer allocated via `snd_pcm_lib_malloc_pages`.

## Return

Zero if successful, or a negative error code on failure.

## Name

`snd_pcm_lib_free_vmalloc_buffer` — free vmalloc buffer

## Synopsis

```
int    snd_pcm_lib_free_vmalloc_buffer    (struct    snd_pcm_substream    *  
substream);
```

## Arguments

*substream* the substream with a buffer allocated by `snd_pcm_lib_alloc_vmalloc_buffer`

## Return

Zero if successful, or a negative error code on failure.

## Name

`snd_pcm_lib_get_vmalloc_page` — map vmalloc buffer offset to page struct

## Synopsis

```
struct page * snd_pcm_lib_get_vmalloc_page (struct snd_pcm_substream *  
substream, unsigned long offset);
```

## Arguments

*substream*    the substream with a buffer allocated by `snd_pcm_lib_alloc_vmalloc_buffer`

*offset*        offset in the buffer

## Description

This function is to be used as the page callback in the PCM ops.

## Return

The page struct, or NULL on failure.

## Name

`snd_device_initialize` — Initialize struct device for sound devices

## Synopsis

```
void snd_device_initialize (struct device * dev, struct snd_card * card);
```

## Arguments

*dev*     device to initialize

*card*    card to assign, optional

## Name

`snd_card_new` — create and initialize a soundcard structure

## Synopsis

```
int snd_card_new (struct device * parent, int idx, const char * xid,  
struct module * module, int extra_size, struct snd_card ** card_ret);
```

## Arguments

<i>parent</i>	the parent device object
<i>idx</i>	card index (address) [0 ... (SNDRV_CARDS-1)]
<i>xid</i>	card identification (ASCII string)
<i>module</i>	top level module for locking
<i>extra_size</i>	allocate this extra size after the main soundcard structure
<i>card_ret</i>	the pointer to store the created card instance

## Description

Creates and initializes a soundcard structure.

The function allocates `snd_card` instance via `kzalloc` with the given space for the driver to use freely. The allocated struct is stored in the given `card_ret` pointer.

## Return

Zero if successful or a negative error code.

## Name

`snd_card_disconnect` — disconnect all APIs from the file-operations (user space)

## Synopsis

```
int snd_card_disconnect (struct snd_card * card);
```

## Arguments

*card*    soundcard structure

## Description

Disconnects all APIs from the file-operations (user space).

## Return

Zero, otherwise a negative error code.

## Note

The current implementation replaces all active `file->f_op` with special dummy file operations (they do nothing except release).

## Name

`snd_card_free_when_closed` — Disconnect the card, free it later eventually

## Synopsis

```
int snd_card_free_when_closed (struct snd_card * card);
```

## Arguments

*card*    soundcard structure

## Description

Unlike `snd_card_free`, this function doesn't try to release the card resource immediately, but tries to disconnect at first. When the card is still in use, the function returns before freeing the resources. The card resources will be freed when the refcount gets to zero.

## Name

`snd_card_free` — frees given soundcard structure

## Synopsis

```
int snd_card_free (struct snd_card * card);
```

## Arguments

*card* soundcard structure

## Description

This function releases the soundcard structure and the all assigned devices automatically. That is, you don't have to release the devices by yourself.

This function waits until the all resources are properly released.

## Return

Zero. Frees all associated devices and frees the control interface associated to given soundcard.



## Name

`snd_card_set_id` — set card identification name

## Synopsis

```
void snd_card_set_id (struct snd_card * card, const char * nid);
```

## Arguments

*card*    soundcard structure

*nid*     new identification string

## Description

This function sets the card identification and checks for name collisions.

## Name

`snd_card_add_dev_attr` — Append a new sysfs attribute group to card

## Synopsis

```
int snd_card_add_dev_attr (struct snd_card * card, const struct
attribute_group * group);
```

## Arguments

*card*    card instance

*group*   attribute group to append

## Name

`snd_card_register` — register the soundcard

## Synopsis

```
int snd_card_register (struct snd_card * card);
```

## Arguments

*card* soundcard structure

## Description

This function registers all the devices assigned to the soundcard. Until calling this, the ALSA control interface is blocked from the external accesses. Thus, you should call this function at the end of the initialization of the card.

## Return

Zero otherwise a negative error code if the registration failed.

## Name

`snd_component_add` — add a component string

## Synopsis

```
int snd_component_add (struct snd_card * card, const char * component);
```

## Arguments

*card*            soundcard structure

*component*    the component id string

## Description

This function adds the component id string to the supported list. The component can be referred from the `alsa-lib`.

## Return

Zero otherwise a negative error code.

## Name

`snd_card_file_add` — add the file to the file list of the card

## Synopsis

```
int snd_card_file_add (struct snd_card * card, struct file * file);
```

## Arguments

*card*    soundcard structure

*file*    file pointer

## Description

This function adds the file to the file linked-list of the card. This linked-list is used to keep tracking the connection state, and to avoid the release of busy resources by hotplug.

## Return

zero or a negative error code.

## Name

`snd_card_file_remove` — remove the file from the file list

## Synopsis

```
int snd_card_file_remove (struct snd_card * card, struct file * file);
```

## Arguments

*card* soundcard structure

*file* file pointer

## Description

This function removes the file formerly added to the card via `snd_card_file_add` function. If all files are removed and `snd_card_free_when_closed` was called beforehand, it processes the pending release of resources.

## Return

Zero or a negative error code.

## Name

`snd_power_wait` — wait until the power-state is changed.

## Synopsis

```
int snd_power_wait (struct snd_card * card, unsigned int power_state);
```

## Arguments

*card*                    soundcard structure

*power\_state*    expected power state

## Description

Waits until the power-state is changed.

## Return

Zero if successful, or a negative error code.

## Note

the power lock must be active before call.

## Name

`snd_dma_program` — program an ISA DMA transfer

## Synopsis

```
void snd_dma_program (unsigned long dma, unsigned long addr, unsigned  
int size, unsigned short mode);
```

## Arguments

*dma*     the dma number

*addr*    the physical address of the buffer

*size*    the DMA transfer size

*mode*    the DMA transfer mode, `DMA_MODE_XXX`

## Description

Programs an ISA DMA transfer for the given buffer.



## Name

`snd_dma_disable` — stop the ISA DMA transfer

## Synopsis

```
void snd_dma_disable (unsigned long dma);
```

## Arguments

*dma* the dma number

## Description

Stops the ISA DMA transfer.

## Name

`snd_dma_pointer` — return the current pointer to DMA transfer buffer in bytes

## Synopsis

```
unsigned int snd_dma_pointer (unsigned long dma, unsigned int size);
```

## Arguments

*dma*     the dma number

*size*    the dma transfer size

## Return

The current pointer in DMA transfer buffer in bytes.

## Name

`snd_ctl_notify` — Send notification to user-space for a control change

## Synopsis

```
void snd_ctl_notify (struct snd_card * card, unsigned int mask, struct  
snd_ctl_elem_id * id);
```

## Arguments

*card* the card to send notification

*mask* the event mask, `SNDRV_CTL_EVENT_*`

*id* the ctl element id to send notification

## Description

This function adds an event record with the given id and mask, appends to the list and wakes up the user-space for notification. This can be called in the atomic context.

## Name

`snd_ctl_new1` — create a control instance from the template

## Synopsis

```
struct snd_kcontrol * snd_ctl_new1 (const struct snd_kcontrol_new *  
ncontrol, void * private_data);
```

## Arguments

*ncontrol*            the initialization record

*private\_data*    the private data to set

## Description

Allocates a new struct `snd_kcontrol` instance and initialize from the given template. When the access field of `ncontrol` is 0, it's assumed as `READWRITE` access. When the count field is 0, it's assumes as one.

## Return

The pointer of the newly generated instance, or `NULL` on failure.

## Name

`snd_ctl_free_one` — release the control instance

## Synopsis

```
void snd_ctl_free_one (struct snd_kcontrol * kcontrol);
```

## Arguments

*kcontrol* the control instance

## Description

Releases the control instance created via `snd_ctl_new` or `snd_ctl_new1`. Don't call this after the control was added to the card.

## Name

`snd_ctl_add` — add the control instance to the card

## Synopsis

```
int snd_ctl_add (struct snd_card * card, struct snd_kcontrol * kcontrol);
```

## Arguments

*card*            the card instance

*kcontrol*       the control instance to add

## Description

Adds the control instance created via `snd_ctl_new` or `snd_ctl_new1` to the given card. Assigns also an unique numid used for fast search.

It frees automatically the control which cannot be added.

## Return

Zero if successful, or a negative error code on failure.

## Name

`snd_ctl_replace` — replace the control instance of the card

## Synopsis

```
int snd_ctl_replace (struct snd_card * card, struct snd_kcontrol *  
kcontrol, bool add_on_replace);
```

## Arguments

<i>card</i>	the card instance
<i>kcontrol</i>	the control instance to replace
<i>add_on_replace</i>	add the control if not already added

## Description

Replaces the given control. If the given control does not exist and the `add_on_replace` flag is set, the control is added. If the control exists, it is destroyed first.

It frees automatically the control which cannot be added or replaced.

## Return

Zero if successful, or a negative error code on failure.

## Name

`snd_ctl_remove` — remove the control from the card and release it

## Synopsis

```
int snd_ctl_remove (struct snd_card * card, struct snd_kcontrol *  
kcontrol);
```

## Arguments

*card*            the card instance

*kcontrol*    the control instance to remove

## Description

Removes the control from the card and then releases the instance. You don't need to call `snd_ctl_free_one`. You must be in the write lock - `down_write(card->controls_rwsem)`.

## Return

0 if successful, or a negative error code on failure.



## Name

`snd_ctl_remove_id` — remove the control of the given id and release it

## Synopsis

```
int snd_ctl_remove_id (struct snd_card * card, struct snd_ctl_elem_id  
* id);
```

## Arguments

*card*    the card instance

*id*      the control id to remove

## Description

Finds the control instance with the given id, removes it from the card list and releases it.

## Return

0 if successful, or a negative error code on failure.

## Name

`snd_ctl_activate_id` — activate/inactivate the control of the given id

## Synopsis

```
int snd_ctl_activate_id (struct snd_card * card, struct snd_ctl_elem_id
* id, int active);
```

## Arguments

*card*      the card instance

*id*        the control id to activate/inactivate

*active*    non-zero to activate

## Description

Finds the control instance with the given id, and activate or inactivate the control together with notification, if changed. The given ID data is filled with full information.

## Return

0 if unchanged, 1 if changed, or a negative error code on failure.

## Name

`snd_ctl_rename_id` — replace the id of a control on the card

## Synopsis

```
int snd_ctl_rename_id (struct snd_card * card, struct snd_ctl_elem_id
* src_id, struct snd_ctl_elem_id * dst_id);
```

## Arguments

*card*      the card instance

*src\_id*    the old id

*dst\_id*    the new id

## Description

Finds the control with the old id from the card, and replaces the id with the new one.

## Return

Zero if successful, or a negative error code on failure.

## Name

`snd_ctl_find_numid` — find the control instance with the given number-id

## Synopsis

```
struct snd_kcontrol * snd_ctl_find_numid (struct snd_card * card,  
unsigned int numid);
```

## Arguments

*card*     the card instance

*numid*    the number-id to search

## Description

Finds the control instance with the given number-id from the card.

The caller must down `card->controls_rwsem` before calling this function (if the race condition can happen).

## Return

The pointer of the instance if found, or `NULL` if not.

## Name

`snd_ctl_find_id` — find the control instance with the given id

## Synopsis

```
struct snd_kcontrol * snd_ctl_find_id (struct snd_card * card, struct
snd_ctl_elem_id * id);
```

## Arguments

*card*    the card instance

*id*      the id to search

## Description

Finds the control instance with the given id from the card.

The caller must down `card->controls_rwsem` before calling this function (if the race condition can happen).

## Return

The pointer of the instance if found, or `NULL` if not.

## Name

`snd_ctl_register_ioctl` — register the device-specific control-ioctls

## Synopsis

```
int snd_ctl_register_ioctl (snd_kctl_ioctl_func_t fcn);
```

## Arguments

*fcn*    ioctl callback function

## Description

called from each device manager like `pcm.c`, `hwdep.c`, etc.

## Name

`snd_ctl_register_ioctl_compat` — register the device-specific 32bit compat control-ioctls

## Synopsis

```
int snd_ctl_register_ioctl_compat (snd_kctl_ioctl_func_t fcn);
```

## Arguments

*fcn*    ioctl callback function

## Name

`snd_ctl_unregister_ioctl` — de-register the device-specific control-ioctl

## Synopsis

```
int snd_ctl_unregister_ioctl (snd_kctl_ioctl_func_t fcn);
```

## Arguments

*fcn*    ioctl callback function to unregister



## Name

`snd_ctl_unregister_ioctl_compat` — de-register the device-specific compat 32bit control-iocls

## Synopsis

```
int snd_ctl_unregister_ioctl_compat (snd_kctl_ioctl_func_t fcn);
```

## Arguments

*fcn*    ioctl callback function to unregister

## Name

`snd_ctl_boolean_mono_info` — Helper function for a standard boolean info callback with a mono channel

## Synopsis

```
int snd_ctl_boolean_mono_info (struct snd_kcontrol * kcontrol, struct
snd_ctl_elem_info * uinfo);
```

## Arguments

*kcontrol*    the kcontrol instance

*uinfo*       info to store

## Description

This is a function that can be used as info callback for a standard boolean control with a single mono channel.

## Name

`snd_ctl_boolean_stereo_info` — Helper function for a standard boolean info callback with stereo two channels

## Synopsis

```
int snd_ctl_boolean_stereo_info (struct snd_kcontrol * kcontrol, struct
snd_ctl_elem_info * uinfo);
```

## Arguments

*kcontrol*    the kcontrol instance

*uinfo*       info to store

## Description

This is a function that can be used as info callback for a standard boolean control with stereo two channels.

## Name

`snd_ctl_enum_info` — fills the info structure for an enumerated control

## Synopsis

```
int snd_ctl_enum_info (struct snd_ctl_elem_info * info, unsigned int
channels, unsigned int items, const char *const names[]);
```

## Arguments

<i>info</i>	the structure to be filled
<i>channels</i>	the number of the control's channels; often one
<i>items</i>	the number of control values; also the size of <i>names</i>
<i>names</i> []	an array containing the names of all control values

## Description

Sets all required fields in *info* to their appropriate values. If the control's accessibility is not the default (readable and writable), the caller has to fill *info->access*.

## Return

Zero.

## Name

`snd_pcm_set_ops` — set the PCM operators

## Synopsis

```
void snd_pcm_set_ops (struct snd_pcm * pcm, int direction, const struct  
snd_pcm_ops * ops);
```

## Arguments

<i>pcm</i>	the pcm instance
<i>direction</i>	stream direction, <code>SNDRV_PCM_STREAM_XXX</code>
<i>ops</i>	the operator table

## Description

Sets the given PCM operators to the pcm instance.

## Name

`snd_pcm_set_sync` — set the PCM sync id

## Synopsis

```
void snd_pcm_set_sync (struct snd_pcm_substream * substream);
```

## Arguments

*substream* the pcm substream

## Description

Sets the PCM sync identifier for the card.

## Name

`snd_interval_refine` — refine the interval value of configurator

## Synopsis

```
int snd_interval_refine (struct snd_interval * i, const struct
snd_interval * v);
```

## Arguments

*i* the interval value to refine

*v* the interval value to refer to

## Description

Refines the interval value with the reference value. The interval is changed to the range satisfying both intervals. The interval status (min, max, integer, etc.) are evaluated.

## Return

Positive if the value is changed, zero if it's not changed, or a negative error code.

## Name

`snd_interval_ratnum` — refine the interval value

## Synopsis

```
int snd_interval_ratnum (struct snd_interval * i, unsigned int
rats_count, const struct snd_ratnum * rats, unsigned int * nump, unsigned
int * denp);
```

## Arguments

<i>i</i>	interval to refine
<i>rats_count</i>	number of <code>ratnum_t</code>
<i>rats</i>	<code>ratnum_t</code> array
<i>nump</i>	pointer to store the resultant numerator
<i>denp</i>	pointer to store the resultant denominator

## Return

Positive if the value is changed, zero if it's not changed, or a negative error code.



## Name

`snd_interval_list` — refine the interval value from the list

## Synopsis

```
int snd_interval_list (struct snd_interval * i, unsigned int count,  
const unsigned int * list, unsigned int mask);
```

## Arguments

*i*            the interval value to refine

*count*      the number of elements in the list

*list*        the value list

*mask*        the bit-mask to evaluate

## Description

Refines the interval value from the list. When mask is non-zero, only the elements corresponding to bit 1 are evaluated.

## Return

Positive if the value is changed, zero if it's not changed, or a negative error code.

## Name

`snd_interval_ranges` — refine the interval value from the list of ranges

## Synopsis

```
int snd_interval_ranges (struct snd_interval * i, unsigned int count,  
const struct snd_interval * ranges, unsigned int mask);
```

## Arguments

<i>i</i>	the interval value to refine
<i>count</i>	the number of elements in the list of ranges
<i>ranges</i>	the ranges list
<i>mask</i>	the bit-mask to evaluate

## Description

Refines the interval value from the list of ranges. When mask is non-zero, only the elements corresponding to bit 1 are evaluated.

## Return

Positive if the value is changed, zero if it's not changed, or a negative error code.

## Name

`snd_pcm_hw_rule_add` — add the hw-constraint rule

## Synopsis

```
int snd_pcm_hw_rule_add (struct snd_pcm_runtime * runtime, unsigned int
cond, int var, snd_pcm_hw_rule_func_t func, void * private, int dep,
...);
```

## Arguments

<i>runtime</i>	the pcm runtime instance
<i>cond</i>	condition bits
<i>var</i>	the variable to evaluate
<i>func</i>	the evaluation function
<i>private</i>	the private data pointer passed to function
<i>dep</i>	the dependent variables
...	variable arguments

## Return

Zero if successful, or a negative error code on failure.

## Name

`snd_pcm_hw_constraint_mask64` — apply the given bitmap mask constraint

## Synopsis

```
int snd_pcm_hw_constraint_mask64 (struct snd_pcm_runtime * runtime,
snd_pcm_hw_param_t var, u_int64_t mask);
```

## Arguments

*runtime*    PCM runtime instance

*var*        hw\_params variable to apply the mask

*mask*       the 64bit bitmap mask

## Description

Apply the constraint of the given bitmap mask to a 64-bit mask parameter.

## Return

Zero if successful, or a negative error code on failure.

## Name

`snd_pcm_hw_constraint_integer` — apply an integer constraint to an interval

## Synopsis

```
int snd_pcm_hw_constraint_integer (struct snd_pcm_runtime * runtime,
snd_pcm_hw_param_t var);
```

## Arguments

*runtime*    PCM runtime instance

*var*        hw\_params variable to apply the integer constraint

## Description

Apply the constraint of integer to an interval parameter.

## Return

Positive if the value is changed, zero if it's not changed, or a negative error code.

## Name

`snd_pcm_hw_constraint_minmax` — apply a min/max range constraint to an interval

## Synopsis

```
int snd_pcm_hw_constraint_minmax (struct snd_pcm_runtime * runtime,
snd_pcm_hw_param_t var, unsigned int min, unsigned int max);
```

## Arguments

<i>runtime</i>	PCM runtime instance
<i>var</i>	hw_params variable to apply the range
<i>min</i>	the minimal value
<i>max</i>	the maximal value

## Description

Apply the min/max range constraint to an interval parameter.

## Return

Positive if the value is changed, zero if it's not changed, or a negative error code.

## Name

`snd_pcm_hw_constraint_list` — apply a list of constraints to a parameter

## Synopsis

```
int  snd_pcm_hw_constraint_list (struct snd_pcm_runtime * runtime,
unsigned int  cond,    snd_pcm_hw_param_t  var,    const struct
snd_pcm_hw_constraint_list * l);
```

## Arguments

*runtime* PCM runtime instance

*cond* condition bits

*var* hw\_params variable to apply the list constraint

*l* list

## Description

Apply the list of constraints to an interval parameter.

## Return

Zero if successful, or a negative error code on failure.

## Name

`snd_pcm_hw_constraint_ranges` — apply list of range constraints to a parameter

## Synopsis

```
int snd_pcm_hw_constraint_ranges (struct snd_pcm_runtime * runtime,
unsigned int cond, snd_pcm_hw_param_t var, const struct
snd_pcm_hw_constraint_ranges * r);
```

## Arguments

<i>runtime</i>	PCM runtime instance
<i>cond</i>	condition bits
<i>var</i>	hw_params variable to apply the list of range constraints
<i>r</i>	ranges

## Description

Apply the list of range constraints to an interval parameter.

## Return

Zero if successful, or a negative error code on failure.



## Name

`snd_pcm_hw_constraint_ratnums` — apply ratnums constraint to a parameter

## Synopsis

```
int snd_pcm_hw_constraint_ratnums (struct snd_pcm_runtime * runtime,
unsigned int cond, snd_pcm_hw_param_t var, const struct
snd_pcm_hw_constraint_ratnums * r);
```

## Arguments

*runtime*    PCM runtime instance

*cond*       condition bits

*var*        hw\_params variable to apply the ratnums constraint

*r*           struct snd\_ratnums constraints

## Return

Zero if successful, or a negative error code on failure.

## Name

`snd_pcm_hw_constraint_ratdens` — apply ratdens constraint to a parameter

## Synopsis

```
int snd_pcm_hw_constraint_ratdens (struct snd_pcm_runtime * runtime,
unsigned int cond, snd_pcm_hw_param_t var, const struct
snd_pcm_hw_constraint_ratdens * r);
```

## Arguments

*runtime*    PCM runtime instance

*cond*       condition bits

*var*        hw\_params variable to apply the ratdens constraint

*r*           struct snd\_ratdens constraints

## Return

Zero if successful, or a negative error code on failure.

## Name

`snd_pcm_hw_constraint_msbits` — add a hw constraint msbits rule

## Synopsis

```
int snd_pcm_hw_constraint_msbits (struct snd_pcm_runtime * runtime,
unsigned int cond, unsigned int width, unsigned int msbits);
```

## Arguments

*runtime*    PCM runtime instance

*cond*       condition bits

*width*      sample bits width

*msbits*     msbits width

## Description

This constraint will set the number of most significant bits (msbits) if a sample format with the specified width has been select. If width is set to 0 the msbits will be set for any sample format with a width larger than the specified msbits.

## Return

Zero if successful, or a negative error code on failure.

## Name

`snd_pcm_hw_constraint_step` — add a hw constraint step rule

## Synopsis

```
int snd_pcm_hw_constraint_step (struct snd_pcm_runtime * runtime,
unsigned int cond, snd_pcm_hw_param_t var, unsigned long step);
```

## Arguments

*runtime*    PCM runtime instance

*cond*       condition bits

*var*        hw\_params variable to apply the step constraint

*step*       step size

## Return

Zero if successful, or a negative error code on failure.

## Name

`snd_pcm_hw_constraint_pow2` — add a hw constraint power-of-2 rule

## Synopsis

```
int  snd_pcm_hw_constraint_pow2 (struct snd_pcm_runtime * runtime,
unsigned int cond, snd_pcm_hw_param_t var);
```

## Arguments

*runtime* PCM runtime instance

*cond* condition bits

*var* hw\_params variable to apply the power-of-2 constraint

## Return

Zero if successful, or a negative error code on failure.

## Name

`snd_pcm_hw_rule_noresample` — add a rule to allow disabling hw resampling

## Synopsis

```
int  snd_pcm_hw_rule_noresample (struct  snd_pcm_runtime  *  runtime,
unsigned int  base_rate);
```

## Arguments

*runtime*     PCM runtime instance

*base\_rate*   the rate at which the hardware does not resample

## Return

Zero if successful, or a negative error code on failure.

## Name

`snd_pcm_hw_param_value` — return *params* field *var* value

## Synopsis

```
int snd_pcm_hw_param_value (const struct snd_pcm_hw_params * params,
snd_pcm_hw_param_t var, int * dir);
```

## Arguments

*params*    the hw\_params instance

*var*       parameter to retrieve

*dir*       pointer to the direction (-1,0,1) or NULL

## Return

The value for field *var* if it's fixed in configuration space defined by *params*. -EINVAL otherwise.

## Name

`snd_pcm_hw_param_first` — refine config space and return minimum value

## Synopsis

```
int snd_pcm_hw_param_first (struct snd_pcm_substream * pcm, struct
snd_pcm_hw_params * params, snd_pcm_hw_param_t var, int * dir);
```

## Arguments

<i>pcm</i>	PCM instance
<i>params</i>	the <code>hw_params</code> instance
<i>var</i>	parameter to retrieve
<i>dir</i>	pointer to the direction (-1,0,1) or NULL

## Description

Inside configuration space defined by *params* remove from *var* all values > minimum. Reduce configuration space accordingly.

## Return

The minimum, or a negative error code on failure.



## Name

`snd_pcm_hw_param_last` — refine config space and return maximum value

## Synopsis

```
int snd_pcm_hw_param_last (struct snd_pcm_substream * pcm, struct
snd_pcm_hw_params * params, snd_pcm_hw_param_t var, int * dir);
```

## Arguments

<i>pcm</i>	PCM instance
<i>params</i>	the <code>hw_params</code> instance
<i>var</i>	parameter to retrieve
<i>dir</i>	pointer to the direction (-1,0,1) or NULL

## Description

Inside configuration space defined by *params* remove from *var* all values < maximum. Reduce configuration space accordingly.

## Return

The maximum, or a negative error code on failure.

## Name

`snd_pcm_lib_ioctl` — a generic PCM ioctl callback

## Synopsis

```
int snd_pcm_lib_ioctl (struct snd_pcm_substream * substream, unsigned
int cmd, void * arg);
```

## Arguments

*substream*    the pcm substream instance

*cmd*            ioctl command

*arg*            ioctl argument

## Description

Processes the generic ioctl commands for PCM. Can be passed as the ioctl callback for PCM ops.

## Return

Zero if successful, or a negative error code on failure.

## Name

`snd_pcm_period_elapsed` — update the pcm status for the next period

## Synopsis

```
void snd_pcm_period_elapsed (struct snd_pcm_substream * substream);
```

## Arguments

*substream* the pcm substream instance

## Description

This function is called from the interrupt handler when the PCM has processed the period size. It will update the current pointer, wake up sleepers, etc.

Even if more than one periods have elapsed since the last call, you have to call this only once.

## Name

`snd_pcm_add_chmap_ctls` — create channel-mapping control elements

## Synopsis

```
int snd_pcm_add_chmap_ctls (struct snd_pcm * pcm, int stream, const
struct snd_pcm_chmap_elem * chmap, int max_channels, unsigned long
private_value, struct snd_pcm_chmap ** info_ret);
```

## Arguments

<i>pcm</i>	the assigned PCM instance
<i>stream</i>	stream direction
<i>chmap</i>	channel map elements (for query)
<i>max_channels</i>	the max number of channels for the stream
<i>private_value</i>	the value passed to each kcontrol's <code>private_value</code> field
<i>info_ret</i>	store struct <code>snd_pcm_chmap</code> instance if non-NULL

## Description

Create channel-mapping control elements assigned to the given PCM stream(s).

## Return

Zero if successful, or a negative error value.

## Name

`snd_hwdep_new` — create a new hwdep instance

## Synopsis

```
int snd_hwdep_new (struct snd_card * card, char * id, int device, struct
snd_hwdep ** rhwdep);
```

## Arguments

*card*      the card instance

*id*        the id string

*device*    the device index (zero-based)

*rhwdep*    the pointer to store the new hwdep instance

## Description

Creates a new hwdep instance with the given index on the card. The callbacks (`hwdep->ops`) must be set on the returned instance after this call manually by the caller.

## Return

Zero if successful, or a negative error code on failure.

## Name

`snd_pcm_stream_lock` — Lock the PCM stream

## Synopsis

```
void snd_pcm_stream_lock (struct snd_pcm_substream * substream);
```

## Arguments

*substream*   PCM substream

## Description

This locks the PCM stream's spinlock or mutex depending on the nonatomic flag of the given substream. This also takes the global link rw lock (or rw sem), too, for avoiding the race with linked streams.

## Name

`snd_pcm_stream_unlock` — Unlock the PCM stream

## Synopsis

```
void snd_pcm_stream_unlock (struct snd_pcm_substream * substream);
```

## Arguments

*substream*   PCM substream

## Description

This unlocks the PCM stream that has been locked via `snd_pcm_stream_lock`.

## Name

`snd_pcm_stream_lock_irq` — Lock the PCM stream

## Synopsis

```
void snd_pcm_stream_lock_irq (struct snd_pcm_substream * substream);
```

## Arguments

*substream*   PCM substream

## Description

This locks the PCM stream like `snd_pcm_stream_lock` and disables the local IRQ (only when `nonatomic` is false). In nonatomic case, this is identical as `snd_pcm_stream_lock`.



## Name

`snd_pcm_stream_unlock_irq` — Unlock the PCM stream

## Synopsis

```
void snd_pcm_stream_unlock_irq (struct snd_pcm_substream * substream);
```

## Arguments

*substream*   PCM substream

## Description

This is a counter-part of `snd_pcm_stream_lock_irq`.

## Name

`snd_pcm_stream_unlock_irqrestore` — Unlock the PCM stream

## Synopsis

```
void snd_pcm_stream_unlock_irqrestore (struct snd_pcm_substream *  
substream, unsigned long flags);
```

## Arguments

*substream*    PCM substream

*flags*        irq flags

## Description

This is a counter-part of `snd_pcm_stream_lock_irqsave`.

## Name

`snd_pcm_stop` — try to stop all running streams in the substream group

## Synopsis

```
int snd_pcm_stop (struct snd_pcm_substream * substream, snd_pcm_state_t
state);
```

## Arguments

*substream*    the PCM substream instance

*state*        PCM state after stopping the stream

## Description

The state of each stream is then changed to the given state unconditionally.

## Return

Zero if successful, or a negative error code.

## Name

`snd_pcm_stop_xrun` — stop the running streams as XRUN

## Synopsis

```
int snd_pcm_stop_xrun (struct snd_pcm_substream * substream);
```

## Arguments

*substream* the PCM substream instance

## Description

This stops the given running substream (and all linked substreams) as XRUN. Unlike `snd_pcm_stop`, this function takes the substream lock by itself.

## Return

Zero if successful, or a negative error code.

## Name

`snd_pcm_suspend` — trigger SUSPEND to all linked streams

## Synopsis

```
int snd_pcm_suspend (struct snd_pcm_substream * substream);
```

## Arguments

*substream* the PCM substream

## Description

After this call, all streams are changed to SUSPENDED state.

## Return

Zero if successful (or *substream* is NULL), or a negative error code.

## Name

`snd_pcm_suspend_all` — trigger SUSPEND to all substreams in the given pcm

## Synopsis

```
int snd_pcm_suspend_all (struct snd_pcm * pcm);
```

## Arguments

*pcm* the PCM instance

## Description

After this call, all streams are changed to SUSPENDED state.

## Return

Zero if successful (or *pcm* is NULL), or a negative error code.

## Name

`snd_pcm_lib_default_mmap` — Default PCM data mmap function

## Synopsis

```
int snd_pcm_lib_default_mmap (struct snd_pcm_substream * substream,  
struct vm_area_struct * area);
```

## Arguments

*substream*    PCM substream

*area*        VMA

## Description

This is the default mmap handler for PCM data. When `mmap_pcm_ops` is `NULL`, this function is invoked implicitly.

## Name

`snd_pcm_lib_mmap_iomem` — Default PCM data mmap function for I/O mem

## Synopsis

```
int snd_pcm_lib_mmap_iomem (struct snd_pcm_substream * substream, struct
vm_area_struct * area);
```

## Arguments

*substream*    PCM substream

*area*            VMA

## Description

When your hardware uses the iomapped pages as the hardware buffer and wants to mmap it, pass this function as `mmap` `pcm_ops`. Note that this is supposed to work only on limited architectures.



## Name

`snd_malloc_pages` — allocate pages with the given size

## Synopsis

```
void * snd_malloc_pages (size_t size, gfp_t gfp_flags);
```

## Arguments

*size*            the size to allocate in bytes

*gfp\_flags*    the allocation conditions, GFP\_XXX

## Description

Allocates the physically contiguous pages with the given size.

## Return

The pointer of the buffer, or NULL if no enough memory.

## Name

`snd_free_pages` — release the pages

## Synopsis

```
void snd_free_pages (void * ptr, size_t size);
```

## Arguments

*ptr*     the buffer pointer to release

*size*    the allocated buffer size

## Description

Releases the buffer allocated via `snd_malloc_pages`.

## Name

`snd_dma_alloc_pages` — allocate the buffer area according to the given type

## Synopsis

```
int snd_dma_alloc_pages (int type, struct device * device, size_t size,  
struct snd_dma_buffer * dmab);
```

## Arguments

*type*      the DMA buffer type

*device*    the device pointer

*size*      the buffer size to allocate

*dmab*      buffer allocation record to store the allocated data

## Description

Calls the memory-allocator function for the corresponding buffer type.

## Return

Zero if the buffer with the given size is allocated successfully, otherwise a negative value on error.

## Name

`snd_dma_alloc_pages_fallback` — allocate the buffer area according to the given type with fallback

## Synopsis

```
int snd_dma_alloc_pages_fallback (int type, struct device * device,  
size_t size, struct snd_dma_buffer * dmab);
```

## Arguments

*type*      the DMA buffer type

*device*    the device pointer

*size*      the buffer size to allocate

*dmab*      buffer allocation record to store the allocated data

## Description

Calls the memory-allocator function for the corresponding buffer type. When no space is left, this function reduces the size and tries to allocate again. The size actually allocated is stored in `res_size` argument.

## Return

Zero if the buffer with the given size is allocated successfully, otherwise a negative value on error.

## Name

`snd_dma_free_pages` — release the allocated buffer

## Synopsis

```
void snd_dma_free_pages (struct snd_dma_buffer * dmab);
```

## Arguments

*dmab* the buffer allocation record to release

## Description

Releases the allocated buffer via `snd_dma_alloc_pages`.

---

# Chapter 6. Media Devices

## Video2Linux devices

## Name

enum tuner\_mode — Mode of the tuner

## Synopsis

```
enum tuner_mode {  
    T_RADIO,  
    T_ANALOG_TV  
};
```

## Constants

T\_RADIO            Tuner core will work in radio mode

T\_ANALOG\_TV      Tuner core will work in analog TV mode

## Description

Older boards only had a single tuner device, but some devices have a separate tuner for radio. In any case, the tuner-core needs to know if the tuner chip(s) will be used in radio mode or analog TV mode, as, on radio mode, frequencies are specified on a different range than on TV mode. This enum is used by the tuner core in order to work with the proper tuner range and eventually use a different tuner chip while in radio mode.

## Name

struct tuner\_setup — setup the tuner chipsets

## Synopsis

```
struct tuner_setup {
    unsigned short addr;
    unsigned int type;
    unsigned int mode_mask;
    void * config;
    int (* tuner_callback) (void *dev, int component, int cmd, int arg);
};
```

## Members

addr	I2C address used to control the tuner device/chipset
type	Type of the tuner, as defined at the TUNER_* macros. Each different tuner model should have a unique identifier.
mode_mask	Mask with the allowed tuner modes: V4L2_TUNER_RADIO, V4L2_TUNER_ANALOG_TV and/or V4L2_TUNER_DIGITAL_TV, describing if the tuner should be used to support Radio, analog TV and/or digital TV.
config	Used to send tuner-specific configuration for complex tuners that require extra parameters to be set. Only a very few tuners require it and its usage on newer tuners should be avoided.
tuner_callback	Some tuners require to call back the bridge driver, in order to do some tasks like rising a GPIO at the bridge chipset, in order to do things like resetting the device.

## Description

Older boards only had a single tuner device. Nowadays multiple tuner devices may be present on a single board. Using TUNER\_SET\_TYPE\_ADDR to pass the tuner\_setup structure it is possible to setup each tuner device in turn.

Since multiple devices may be present it is no longer sufficient to send a command to a single i2c device. Instead you should broadcast the command to all i2c devices.

By setting the mode\_mask correctly you can select which commands are accepted by a specific tuner device. For example, set mode\_mask to T\_RADIO if the device is a radio-only tuner. That specific tuner will only accept commands when the tuner is in radio mode and ignore them when the tuner is set to TV mode.



## Name

enum param\_type — type of the tuner parameters

## Synopsis

```
enum param_type {  
    TUNER_PARAM_TYPE_RADIO,  
    TUNER_PARAM_TYPE_PAL,  
    TUNER_PARAM_TYPE_SECAM,  
    TUNER_PARAM_TYPE_NTSC,  
    TUNER_PARAM_TYPE_DIGITAL  
};
```

## Constants

TUNER_PARAM_TYPE_RADIO	Tuner params are for FM and/or AM radio
TUNER_PARAM_TYPE_PAL	Tuner params are for PAL color TV standard
TUNER_PARAM_TYPE_SECAM	Tuner params are for SECAM color TV standard
TUNER_PARAM_TYPE_NTSC	Tuner params are for NTSC color TV standard
TUNER_PARAM_TYPE_DIGITAL	Tuner params are for digital TV

## Name

struct tuner\_range — define the frequencies supported by the tuner

## Synopsis

```
struct tuner_range {  
    unsigned short limit;  
    unsigned char config;  
    unsigned char cb;  
};
```

## Members

limit	Max frequency supported by that range, in 62.5 kHz (TV) or 62.5 Hz (Radio), as defined by V4L2_TUNER_CAP_LOW.
config	Value of the band switch byte (BB) to setup this mode.
cb	Value of the CB byte to setup this mode.

## Description

Please notice that digital tuners like xc3028/xc4000/xc5000 don't use those ranges, as they're defined inside the driver. This is used by analog tuners that are compatible with the “Philips way” to setup the tuners. On those devices, the tuner set is done via 4 bytes: divider byte1 (DB1), divider byte 2 (DB2), Control byte (CB) and band switch byte (BB). Some tuners also have an additional optional Auxiliary byte (AB).

## Name

struct tuner\_params — Parameters to be used to setup the tuner. Those are used by drivers/media/tuners/tuner-types.c in order to specify the tuner properties. Most of the parameters are for tuners based on tda9887 IF-PLL multi-standard analog TV/Radio demodulator, with is very common on legacy analog tuners.

## Synopsis

```
struct tuner_params {
    enum param_type type;
    unsigned int cb_first_if_lower_freq:1;
    unsigned int has_tda9887:1;
    unsigned int port1_fm_high_sensitivity:1;
    unsigned int port2_fm_high_sensitivity:1;
    unsigned int fm_gain_normal:1;
    unsigned int intercarrier_mode:1;
    unsigned int port1_active:1;
    unsigned int port2_active:1;
    unsigned int port1_invert_for_secam_lc:1;
    unsigned int port2_invert_for_secam_lc:1;
    unsigned int port1_set_for_fm_mono:1;
    unsigned int default_pll_gating_18:1;
    unsigned int radio_if:2;
    signed int default_top_low:5;
    signed int default_top_mid:5;
    signed int default_top_high:5;
    signed int default_top_secam_low:5;
    signed int default_top_secam_mid:5;
    signed int default_top_secam_high:5;
    ul6 iffreq;
    unsigned int count;
    struct tuner_range * ranges;
};
```

## Members

type	Type of the tuner parameters, as defined at enum param_type. If the tuner supports multiple standards, an array should be used, with one row per different standard.
cb_first_if_lower_freq	Many Philips-based tuners have a comment in their datasheet like “For channel selection involving band switching, and to ensure smooth tuning to the desired channel without causing unnecessary charge pump action, it is recommended to consider the difference between wanted channel frequency and the current channel frequency. Unnecessary charge pump action will result in very low tuning voltage which may drive the oscillator to extreme conditions”. Set cb_first_if_lower_freq to 1, if this check is required for this tuner. I tested this for PAL by first setting the TV frequency to 203 MHz and then switching to 96.6 MHz FM radio. The result was static unless the control byte was sent first.
has_tda9887	Set to 1 if this tuner uses a tda9887

port1_fm_high_sensitivity	Many Philips tuners use tda9887 PORT1 to select the FM radio sensitivity. If this setting is 1, then set PORT1 to 1 to get proper FM reception.
port2_fm_high_sensitivity	Some Philips tuners use tda9887 PORT2 to select the FM radio sensitivity. If this setting is 1, then set PORT2 to 1 to get proper FM reception.
fm_gain_normal	Some Philips tuners use tda9887 cGainNormal to select the FM radio sensitivity. If this setting is 1, e register will use cGainNormal instead of cGainLow.
intercarrier_mode	Most tuners with a tda9887 use QSS mode. Some (cheaper) tuners use Intercarrier mode. If this setting is 1, then the tuner needs to be set to intercarrier mode.
port1_active	This setting sets the default value for PORT1. 0 means inactive, 1 means active. Note: the actual bit value written to the tda9887 is inverted. So a 0 here means a 1 in the B6 bit.
port2_active	This setting sets the default value for PORT2. 0 means inactive, 1 means active. Note: the actual bit value written to the tda9887 is inverted. So a 0 here means a 1 in the B7 bit.
port1_invert_for_secam_lc	Sometimes PORT1 is inverted when the SECAM-L' standard is selected. Set this bit to 1 if this is needed.
port2_invert_for_secam_lc	Sometimes PORT2 is inverted when the SECAM-L' standard is selected. Set this bit to 1 if this is needed.
port1_set_for_fm_mono	Some cards require PORT1 to be 1 for mono Radio FM and 0 for stereo.
default_pll_gating_18	Select 18% (or according to datasheet 0%) L standard PLL gating, vs the driver default of 36%.
radio_if	IF to use in radio mode. Tuners with a separate radio IF filter seem to use 10.7, while those without use 33.3 for PAL/SECAM tuners and 41.3 for NTSC tuners. 0 = 10.7, 1 = 33.3, 2 = 41.3
default_top_low	Default tda9887 TOP value in dB for the low band. Default is 0. Range: -16:+15
default_top_mid	Default tda9887 TOP value in dB for the mid band. Default is 0. Range: -16:+15
default_top_high	Default tda9887 TOP value in dB for the high band. Default is 0. Range: -16:+15
default_top_secam_low	Default tda9887 TOP value in dB for SECAM-L/L' for the low band. Default is 0. Several tuners require a different TOP value for the SECAM-L/L' standards. Range: -16:+15
default_top_secam_mid	Default tda9887 TOP value in dB for SECAM-L/L' for the mid band. Default is 0. Several tuners require a different TOP value for the SECAM-L/L' standards. Range: -16:+15

default_top_secam_high	Default tda9887 TOP value in dB for SECAM-L/L' for the high band. Default is 0. Several tuners require a different TOP value for the SECAM-L/L' standards. Range: -16:+15
iffreq	Intermediate frequency (IF) used by the tuner on digital mode.
count	Size of the ranges array.
ranges	Array with the frequency ranges supported by the tuner.

## Name

enum tveeprom\_audio\_processor — Specifies the type of audio processor used on a Hauppauge device.

## Synopsis

```
enum tveeprom_audio_processor {  
    TVEEPROM_AUDPROC_NONE,  
    TVEEPROM_AUDPROC_INTERNAL,  
    TVEEPROM_AUDPROC_MSP,  
    TVEEPROM_AUDPROC_OTHER  
};
```

## Constants

TVEEPROM\_AUDPROC\_NONE    No audio processor present

TVEEPROM\_AUDPROC\_INTERNAL    The audio processor is internal to the video processor

TVEEPROM\_AUDPROC\_MSP    The audio processor is a MSPXXXX device

TVEEPROM\_AUDPROC\_OTHER    The audio processor is another device

## Name

struct tveeprom — Contains the fields parsed from Hauppauge eeproms

## Synopsis

```
struct tveeprom {
    u32 has_radio;
    u32 has_ir;
    u32 has_MAC_address;
    u32 tuner_type;
    u32 tuner_formats;
    u32 tuner_hauppauge_model;
    u32 tuner2_type;
    u32 tuner2_formats;
    u32 tuner2_hauppauge_model;
    u32 audio_processor;
    u32 decoder_processor;
    u32 model;
    u32 revision;
    u32 serial_number;
    char rev_str[5];
    u8 MAC_address[ETH_ALEN];
};
```

## Members

has_radio	1 if the device has radio; 0 otherwise.
has_ir	If has_ir == 0, then it is unknown what the IR capabilities are. Otherwise: bit 0) 1 (= IR capabilities are known); bit 1) IR receiver present; bit 2) IR transmitter (blaster) present.
has_MAC_address	0: no MAC, 1: MAC present, 2: unknown.
tuner_type	type of the tuner (TUNER_*, as defined at include/media/tuner.h).
tuner_formats	Supported analog TV standards (V4L2_STD_*).
tuner_hauppauge_model	Hauppauge's code for the device model number.
tuner2_type	type of the second tuner (TUNER_*, as defined at include/media/tuner.h).
tuner2_formats	Tuner 2 supported analog TV standards (V4L2_STD_*).
tuner2_hauppauge_model	tuner 2 Hauppauge's code for the device model number.
audio_processor	analog audio decoder, as defined by enum tveeprom_audio_processor.
decoder_processor	Hauppauge's code for the decoder chipset. Unused by the drivers, as they probe the decoder based on the PCI or USB ID.
model	Hauppauge's model number
revision	Card revision number

serial_number	Card's serial number
rev_str[5]	Card revision converted to number
MAC_address[ETH_ALEN]	MAC address for the network interface



## Name

`tveeprom_hauppauge_analog` — Fill struct `tveeprom` using the contents of the eeprom previously filled at `eeprom_data` field.

## Synopsis

```
void tveeprom_hauppauge_analog (struct i2c_client * c, struct tveeprom  
* tvee, unsigned char * eeprom_data);
```

## Arguments

<i>c</i>	I2C client struct
<i>tvee</i>	Struct to where the eeprom parsed data will be filled;
<i>eeprom_data</i>	Array with the contents of the <code>eeprom_data</code> . It should contain 256 bytes filled with the contents of the eeprom read from the Hauppauge device.

## Name

`tveeprom_read` — Reads the contents of the eeprom found at the Hauppauge devices.

## Synopsis

```
int tveeprom_read (struct i2c_client * c, unsigned char * eedata, int  
len);
```

## Arguments

*c*            I2C client struct

*eedata*      Array where the eeprom content will be stored.

*len*         Size of *eedata* array. If the eeprom content will be latter be parsed by `tveeprom_hauppauge_analog`, *len* should be, at least, 256.

## Name

struct v4l2\_async\_subdev — sub-device descriptor, as known to a bridge

## Synopsis

```
struct v4l2_async_subdev {
    enum v4l2_async_match_type match_type;
    union match;
    struct list_head list;
};
```

## Members

match_type	type of match that will be used
match	union of per-bus type matching data sets
list	used to link struct v4l2_async_subdev objects, waiting to be probed, to a notifier->waiting list

## Name

struct v4l2\_async\_notifier — v4l2\_device notifier data

## Synopsis

```
struct v4l2_async_notifier {
    unsigned int num_subdevs;
    struct v4l2_async_subdev ** subdevs;
    struct v4l2_device * v4l2_dev;
    struct list_head waiting;
    struct list_head done;
    struct list_head list;
    int (* bound) (struct v4l2_async_notifier *notifier, struct v4l2_subdev *subdev, s
    int (* complete) (struct v4l2_async_notifier *notifier);
    void (* unbind) (struct v4l2_async_notifier *notifier, struct v4l2_subdev *subdev
};
```

## Members

num_subdevs	number of subdevices
subdevs	array of pointers to subdevice descriptors
v4l2_dev	pointer to struct v4l2_device
waiting	list of struct v4l2_async_subdev, waiting for their drivers
done	list of struct v4l2_subdev, already probed
list	member in a global list of notifiers
bound	a subdevice driver has successfully probed one of subdevices
complete	all subdevices have been probed successfully
unbind	a subdevice is leaving

## Name

`union v4l2_ctrl_ptr` — A pointer to a control value.

## Synopsis

```
union v4l2_ctrl_ptr {  
    s32 * p_s32;  
    s64 * p_s64;  
    u8 * p_u8;  
    u16 * p_u16;  
    u32 * p_u32;  
    char * p_char;  
    void * p;  
};
```

## Members

<code>p_s32</code>	Pointer to a 32-bit signed value.
<code>p_s64</code>	Pointer to a 64-bit signed value.
<code>p_u8</code>	Pointer to a 8-bit unsigned value.
<code>p_u16</code>	Pointer to a 16-bit unsigned value.
<code>p_u32</code>	Pointer to a 32-bit unsigned value.
<code>p_char</code>	Pointer to a string.
<code>p</code>	Pointer to a compound value.

## Name

struct v4l2\_ctrl\_ops — The control operations that the driver has to provide.

## Synopsis

```
struct v4l2_ctrl_ops {  
    int (* g_volatile_ctrl) (struct v4l2_ctrl *ctrl);  
    int (* try_ctrl) (struct v4l2_ctrl *ctrl);  
    int (* s_ctrl) (struct v4l2_ctrl *ctrl);  
};
```

## Members

<code>g_volatile_ctrl</code>	Get a new value for this control. Generally only relevant for volatile (and usually read-only) controls such as a control that returns the current signal strength which changes continuously. If not set, then the currently cached value will be returned.
<code>try_ctrl</code>	Test whether the control's value is valid. Only relevant when the usual min/max/step checks are not sufficient.
<code>s_ctrl</code>	Actually set the new control value. <code>s_ctrl</code> is compulsory. The <code>ctrl-&gt;handler-&gt;lock</code> is held when these ops are called, so no one else can access controls owned by that handler.

## Name

struct v4l2\_ctrl\_type\_ops — The control type operations that the driver has to provide.

## Synopsis

```
struct v4l2_ctrl_type_ops {  
    bool (* equal) (const struct v4l2_ctrl *ctrl, u32 idx, union v4l2_ctrl_ptr ptr1, u32 idx2, union v4l2_ctrl_ptr ptr2);  
    void (* init) (const struct v4l2_ctrl *ctrl, u32 idx, union v4l2_ctrl_ptr ptr);  
    void (* log) (const struct v4l2_ctrl *ctrl);  
    int (* validate) (const struct v4l2_ctrl *ctrl, u32 idx, union v4l2_ctrl_ptr ptr);  
};
```

## Members

equal	return true if both values are equal.
init	initialize the value.
log	log the value.
validate	validate the value. Return 0 on success and a negative value otherwise.

## Name

struct v4l2\_ctrl — The control structure.

## Synopsis

```
struct v4l2_ctrl {
    struct list_head node;
    struct list_head ev_subs;
    struct v4l2_ctrl_handler * handler;
    struct v4l2_ctrl ** cluster;
    unsigned ncontrols;
    unsigned int done:1;
    unsigned int is_new:1;
    unsigned int has_changed:1;
    unsigned int is_private:1;
    unsigned int is_auto:1;
    unsigned int is_int:1;
    unsigned int is_string:1;
    unsigned int is_ptr:1;
    unsigned int is_array:1;
    unsigned int has_volatiles:1;
    unsigned int call_notify:1;
    unsigned int manual_mode_value:8;
    const struct v4l2_ctrl_ops * ops;
    const struct v4l2_ctrl_type_ops * type_ops;
    u32 id;
    const char * name;
    enum v4l2_ctrl_type type;
    s64 minimum;
    s64 maximum;
    s64 default_value;
    u32 elems;
    u32 elem_size;
    u32 dims[V4L2_CTRL_MAX_DIMS];
    u32 nr_of_dims;
    union cur;
    union v4l2_ctrl_ptr p_new;
    union v4l2_ctrl_ptr p_cur;
};
```

## Members

node	The list node.
ev_subs	The list of control event subscriptions.
handler	The handler that owns the control.
cluster	Point to start of cluster array.
ncontrols	Number of controls in cluster array.
done	Internal flag: set for each processed control.



is_new	Set when the user specified a new value for this control. It is also set when called from <code>v4l2_ctrl_handler_setup</code> . Drivers should never set this flag.
has_changed	Set when the current value differs from the new value. Drivers should never use this flag.
is_private	If set, then this control is private to its handler and it will not be added to any other handlers. Drivers can set this flag.
is_auto	If set, then this control selects whether the other cluster members are in 'automatic' mode or 'manual' mode. This is used for autogain/gain type clusters. Drivers should never set this flag directly.
is_int	If set, then this control has a simple integer value (i.e. it uses <code>ctrl-&gt;val</code> ).
is_string	If set, then this control has type <code>V4L2_CTRL_TYPE_STRING</code> .
is_ptr	If set, then this control is an array and/or has type <code>&gt;= V4L2_CTRL_COMPOUND_TYPES</code> and/or has type <code>V4L2_CTRL_TYPE_STRING</code> . In other words, struct <code>v4l2_ext_control</code> uses field <code>p</code> to point to the data.
is_array	If set, then this control contains an N-dimensional array.
has_volatiles	If set, then one or more members of the cluster are volatile. Drivers should never touch this flag.
call_notify	If set, then call the handler's notify function whenever the control's value changes.
manual_mode_value	If the <code>is_auto</code> flag is set, then this is the value of the auto control that determines if that control is in manual mode. So if the value of the auto control equals this value, then the whole cluster is in manual mode. Drivers should never set this flag directly.
ops	The control ops.
type_ops	The control type ops.
id	The control ID.
name	The control name.
type	The control type.
minimum	The control's minimum value.
maximum	The control's maximum value.
default_value	The control's default value.
elems	The number of elements in the N-dimensional array.
elem_size	The size in bytes of the control.
dims[V4L2_CTRL_MAX_DIMS]	The size of each dimension.

<code>nr_of_dims</code>	The number of dimensions in <i>dims</i> .
<code>cur</code>	The control's current value.
<code>p_new</code>	The control's new value represented via an union with provides a standard way of accessing control types through a pointer.
<code>p_cur</code>	The control's current value represented via an union with provides a standard way of accessing control types through a pointer.

## Name

struct v4l2\_ctrl\_ref — The control reference.

## Synopsis

```
struct v4l2_ctrl_ref {
    struct list_head node;
    struct v4l2_ctrl_ref * next;
    struct v4l2_ctrl * ctrl;
    struct v4l2_ctrl_helper * helper;
};
```

## Members

node	List node for the sorted list.
next	Single-link list node for the hash.
ctrl	The actual control information.
helper	Pointer to helper struct. Used internally in <code>prepare_ext_ctrls</code> .

## Description

Each control handler has a list of these refs. The `list_head` is used to keep a sorted-by-control-ID list of all controls, while the `next` pointer is used to link the control in the hash's bucket.

## Name

struct v4l2\_ctrl\_handler — The control handler keeps track of all the

## Synopsis

```
struct v4l2_ctrl_handler {
    struct mutex _lock;
    struct mutex * lock;
    struct list_head ctrls;
    struct list_head ctrl_refs;
    struct v4l2_ctrl_ref * cached;
    struct v4l2_ctrl_ref ** buckets;
    v4l2_ctrl_notify_fnc notify;
    void * notify_priv;
    ul6 nr_of_buckets;
    int error;
};
```

## Members

<code>_lock</code>	Default for “lock”.
<code>lock</code>	Lock to control access to this handler and its controls. May be replaced by the user right after init.
<code>ctrls</code>	The list of controls owned by this handler.
<code>ctrl_refs</code>	The list of control references.
<code>cached</code>	The last found control reference. It is common that the same control is needed multiple times, so this is a simple optimization.
<code>buckets</code>	Buckets for the hashing. Allows for quick control lookup.
<code>notify</code>	A notify callback that is called whenever the control changes value. Note that the handler's lock is held when the notify function is called!
<code>notify_priv</code>	Passed as argument to the v4l2_ctrl notify callback.
<code>nr_of_buckets</code>	Total number of buckets in the array.
<code>error</code>	The error code of the first failed control addition.

## controls

both the controls owned by the handler and those inherited from other handlers.

## Name

struct v4l2\_ctrl\_config — Control configuration structure.

## Synopsis

```
struct v4l2_ctrl_config {
    const struct v4l2_ctrl_ops * ops;
    const struct v4l2_ctrl_type_ops * type_ops;
    u32 id;
    const char * name;
    enum v4l2_ctrl_type type;
    s64 min;
    s64 max;
    u64 step;
    s64 def;
    u32 dims[V4L2_CTRL_MAX_DIMS];
    u32 elem_size;
    u32 flags;
    u64 menu_skip_mask;
    const char * const * qmenu;
    const s64 * qmenu_int;
    unsigned int is_private:1;
};
```

## Members

ops	The control ops.
type_ops	The control type ops. Only needed for compound controls.
id	The control ID.
name	The control name.
type	The control type.
min	The control's minimum value.
max	The control's maximum value.
step	The control's step value for non-menu controls.
def	The control's default value.
dims[V4L2_CTRL_MAX_DIMS]	The size of each dimension.
elem_size	The size in bytes of the control.
flags	The control's flags.
menu_skip_mask	The control's skip mask for menu controls. This makes it easy to skip menu items that are not valid. If bit X is set, then menu item X is skipped. Of course, this only works for menus with <= 64 menu items. There are no menus that come close to that number, so this is

OK. Should we ever need more, then this will have to be extended to a bit array.

qmenu

A const char \* array for all menu items. Array entries that are empty strings ("") correspond to non-existing menu items (this is in addition to the menu\_skip\_mask above). The last entry must be NULL.

qmenu\_int

A const s64 integer array for all menu items of the type V4L2\_CTRL\_TYPE\_INTEGER\_MENU.

is\_private

If set, then this control is private to its handler and it will not be added to any other handlers.

## Name

`v4l2_ctrl_handler_init_class` — Initialize the control handler.

## Synopsis

```
int  v4l2_ctrl_handler_init_class (struct v4l2_ctrl_handler * hdl,
unsigned nr_of_controls_hint, struct lock_class_key * key, const char
* name);
```

## Arguments

<i>hdl</i>	The control handler.
<i>nr_of_controls_hint</i>	A hint of how many controls this handler is expected to refer to. This is the total number, so including any inherited controls. It doesn't have to be precise, but if it is way off, then you either waste memory (too many buckets are allocated) or the control lookup becomes slower (not enough buckets are allocated, so there are more slow list lookups). It will always work, though.
<i>key</i>	Used by the lock validator if CONFIG_LOCKDEP is set.
<i>name</i>	Used by the lock validator if CONFIG_LOCKDEP is set.

## Description

Returns an error if the buckets could not be allocated. This error will also be stored in *hdl->error*.

Never use this call directly, always use the `v4l2_ctrl_handler_init` macro that hides the *key* and *name* arguments.

## Name

`v4l2_ctrl_handler_free` — Free all controls owned by the handler and free the control list.

## Synopsis

```
void v4l2_ctrl_handler_free (struct v4l2_ctrl_handler * hdl);
```

## Arguments

*hdl*    The control handler.

## Description

Does nothing if *hdl* == NULL.



## Name

`v4l2_ctrl_lock` — Helper function to lock the handler associated with the control.

## Synopsis

```
void v4l2_ctrl_lock (struct v4l2_ctrl * ctrl);
```

## Arguments

*ctrl*    The control to lock.

## Name

`v4l2_ctrl_unlock` — Helper function to unlock the handler associated with the control.

## Synopsis

```
void v4l2_ctrl_unlock (struct v4l2_ctrl * ctrl);
```

## Arguments

*ctrl*    The control to unlock.

## Name

`v4l2_ctrl_handler_setup` — Call the `s_ctrl` op for all controls belonging to the handler to initialize the hardware to the current control values.

## Synopsis

```
int v4l2_ctrl_handler_setup (struct v4l2_ctrl_handler * hdl);
```

## Arguments

*hdl*    The control handler.

## Description

Button controls will be skipped, as are read-only controls.

If *hdl* == NULL, then this just returns 0.

## Name

`v4l2_ctrl_handler_log_status` — Log all controls owned by the handler.

## Synopsis

```
void v4l2_ctrl_handler_log_status (struct v4l2_ctrl_handler * hdl, const  
char * prefix);
```

## Arguments

*hdl*        The control handler.

*prefix*    The prefix to use when logging the control values. If the prefix does not end with a space, then “: ” will be added after the prefix. If *prefix* == NULL, then no prefix will be used.

## Description

For use with VIDIOC\_LOG\_STATUS.

Does nothing if *hdl* == NULL.

## Name

`v4l2_ctrl_new_custom` — Allocate and initialize a new custom V4L2 control.

## Synopsis

```
struct v4l2_ctrl * v4l2_ctrl_new_custom (struct v4l2_ctrl_handler * hdl,  
const struct v4l2_ctrl_config * cfg, void * priv);
```

## Arguments

*hdl*    The control handler.

*cfg*    The control's configuration data.

*priv*   The control's driver-specific private data.

## Description

If the `v4l2_ctrl` struct could not be allocated then `NULL` is returned and *hdl*->error is set to the error code (if it wasn't set already).

## Name

`v4l2_ctrl_new_std` — Allocate and initialize a new standard V4L2 non-menu control.

## Synopsis

```
struct v4l2_ctrl * v4l2_ctrl_new_std (struct v4l2_ctrl_handler * hdl,
const struct v4l2_ctrl_ops * ops, u32 id, s64 min, s64 max, u64 step,
s64 def);
```

## Arguments

*hdl*    The control handler.

*ops*    The control ops.

*id*    The control ID.

*min*    The control's minimum value.

*max*    The control's maximum value.

*step*   The control's step value

*def*    The control's default value.

## Description

If the `v4l2_ctrl` struct could not be allocated, or the control ID is not known, then NULL is returned and `hdl->error` is set to the appropriate error code (if it wasn't set already).

If *id* refers to a menu control, then this function will return NULL.

Use `v4l2_ctrl_new_std_menu` when adding menu controls.

## Name

`v4l2_ctrl_new_std_menu` — Allocate and initialize a new standard V4L2 menu control.

## Synopsis

```
struct v4l2_ctrl * v4l2_ctrl_new_std_menu (struct v4l2_ctrl_handler *  
hdl, const struct v4l2_ctrl_ops * ops, u32 id, u8 max, u64 mask, u8 def);
```

## Arguments

*hdl*     The control handler.

*ops*     The control ops.

*id*      The control ID.

*max*     The control's maximum value.

*mask*    The control's skip mask for menu controls. This makes it easy to skip menu items that are not valid. If bit X is set, then menu item X is skipped. Of course, this only works for menus with  $\leq 64$  menu items. There are no menus that come close to that number, so this is OK. Should we ever need more, then this will have to be extended to a bit array.

*def*     The control's default value.

## Description

Same as `v4l2_ctrl_new_std`, but *min* is set to 0 and the *mask* value determines which menu items are to be skipped.

If *id* refers to a non-menu control, then this function will return NULL.

## Name

`v4l2_ctrl_new_std_menu_items` — Create a new standard V4L2 menu control with driver specific menu.

## Synopsis

```
struct v4l2_ctrl * v4l2_ctrl_new_std_menu_items (struct v4l2_ctrl_handler * hdl, const struct v4l2_ctrl_ops * ops, u32 id, u8 max, u64 mask, u8 def, const char *const * qmenu);
```

## Arguments

<i>hdl</i>	The control handler.
<i>ops</i>	The control ops.
<i>id</i>	The control ID.
<i>max</i>	The control's maximum value.
<i>mask</i>	The control's skip mask for menu controls. This makes it easy to skip menu items that are not valid. If bit X is set, then menu item X is skipped. Of course, this only works for menus with <= 64 menu items. There are no menus that come close to that number, so this is OK. Should we ever need more, then this will have to be extended to a bit array.
<i>def</i>	The control's default value.
<i>qmenu</i>	The new menu.

## Description

Same as `v4l2_ctrl_new_std_menu`, but *qmenu* will be the driver specific menu of this control.



## Name

`v4l2_ctrl_new_int_menu` — Create a new standard V4L2 integer menu control.

## Synopsis

```
struct v4l2_ctrl * v4l2_ctrl_new_int_menu (struct v4l2_ctrl_handler *  
hdl, const struct v4l2_ctrl_ops * ops, u32 id, u8 max, u8 def, const  
s64 * qmenu_int);
```

## Arguments

<i>hdl</i>	The control handler.
<i>ops</i>	The control ops.
<i>id</i>	The control ID.
<i>max</i>	The control's maximum value.
<i>def</i>	The control's default value.
<i>qmenu_int</i>	The control's menu entries.

## Description

Same as `v4l2_ctrl_new_std_menu`, but *mask* is set to 0 and it additionally takes as an argument an array of integers determining the menu items.

If *id* refers to a non-integer-menu control, then this function will return NULL.

## Name

`v4l2_ctrl_add_ctrl` — Add a control from another handler to this handler.

## Synopsis

```
struct v4l2_ctrl * v4l2_ctrl_add_ctrl (struct v4l2_ctrl_handler * hdl,  
struct v4l2_ctrl * ctrl);
```

## Arguments

*hdl*    The control handler.

*ctrl*   The control to add.

## Description

It will return NULL if it was unable to add the control reference. If the control already belonged to the handler, then it will do nothing and just return *ctrl*.

## Name

`v4l2_ctrl_add_handler` — Add all controls from handler *add* to handler *hdl*.

## Synopsis

```
int v4l2_ctrl_add_handler (struct v4l2_ctrl_handler * hdl, struct
v4l2_ctrl_handler * add, bool (*filter) (const struct v4l2_ctrl *ctrl));
```

## Arguments

*hdl*        The control handler.

*add*        The control handler whose controls you want to add to the *hdl* control handler.

*filter*    This function will filter which controls should be added.

## Description

Does nothing if either of the two handlers is a NULL pointer. If *filter* is NULL, then all controls are added. Otherwise only those controls for which *filter* returns true will be added. In case of an error *hdl->error* will be set to the error code (if it wasn't set already).

## Name

`v4l2_ctrl_radio_filter` — Standard filter for radio controls.

## Synopsis

```
bool v4l2_ctrl_radio_filter (const struct v4l2_ctrl * ctrl);
```

## Arguments

*ctrl* The control that is filtered.

## Description

This will return true for any controls that are valid for radio device nodes. Those are all of the `V4L2_CID_AUDIO_*` user controls and all FM transmitter class controls.

This function is to be used with `v4l2_ctrl_add_handler`.

## Name

`v4l2_ctrl_cluster` — Mark all controls in the cluster as belonging to that cluster.

## Synopsis

```
void v4l2_ctrl_cluster (unsigned ncontrols, struct v4l2_ctrl **  
controls);
```

## Arguments

*ncontrols*    The number of controls in this cluster.

*controls*    The cluster control array of size *ncontrols*.

## Name

`v4l2_ctrl_auto_cluster` — Mark all controls in the cluster as belonging to that cluster and set it up for autofoo/foo-type handling.

## Synopsis

```
void v4l2_ctrl_auto_cluster (unsigned ncontrols, struct v4l2_ctrl **  
controls, u8 manual_val, bool set_volatile);
```

## Arguments

<i>ncontrols</i>	The number of controls in this cluster.
<i>controls</i>	The cluster control array of size <i>ncontrols</i> . The first control must be the 'auto' control (e.g. autogain, autoexposure, etc.)
<i>manual_val</i>	The value for the first control in the cluster that equals the manual setting.
<i>set_volatile</i>	If true, then all controls except the first auto control will be volatile.

## Description

Use for control groups where one control selects some automatic feature and the other controls are only active whenever the automatic feature is turned off (manual mode). Typical examples: autogain vs gain, auto-whitebalance vs red and blue balance, etc.

## The behavior of such controls is as follows

When the autofoo control is set to automatic, then any manual controls are set to inactive and any reads will call `g_volatile_ctrl` (if the control was marked volatile).

When the autofoo control is set to manual, then any manual controls will be marked active, and any reads will just return the current value without going through `g_volatile_ctrl`.

In addition, this function will set the `V4L2_CTRL_FLAG_UPDATE` flag on the autofoo control and `V4L2_CTRL_FLAG_INACTIVE` on the foo control(s) if autofoo is in auto mode.

## Name

`v4l2_ctrl_find` — Find a control with the given ID.

## Synopsis

```
struct v4l2_ctrl * v4l2_ctrl_find (struct v4l2_ctrl_handler * hdl, u32
id);
```

## Arguments

*hdl*    The control handler.

*id*     The control ID to find.

## Description

If *hdl* == NULL this will return NULL as well. Will lock the handler so do not use from inside `v4l2_ctrl_ops`.

## Name

`v4l2_ctrl_activate` — Make the control active or inactive.

## Synopsis

```
void v4l2_ctrl_activate (struct v4l2_ctrl * ctrl, bool active);
```

## Arguments

*ctrl*      The control to (de)activate.

*active*    True if the control should become active.

## Description

This sets or clears the `V4L2_CTRL_FLAG_INACTIVE` flag atomically. Does nothing if *ctrl* == `NULL`. This will usually be called from within the `s_ctrl` op. The `V4L2_EVENT_CTRL` event will be generated afterwards.

This function assumes that the control handler is locked.



## Name

`v4l2_ctrl_grab` — Mark the control as grabbed or not grabbed.

## Synopsis

```
void v4l2_ctrl_grab (struct v4l2_ctrl * ctrl, bool grabbed);
```

## Arguments

*ctrl*        The control to (de)activate.

*grabbed*    True if the control should become grabbed.

## Description

This sets or clears the `V4L2_CTRL_FLAG_GRABBED` flag atomically. Does nothing if `ctrl == NULL`. The `V4L2_EVENT_CTRL` event will be generated afterwards. This will usually be called when starting or stopping streaming in the driver.

This function assumes that the control handler is not locked and will take the lock itself.

## Name

`__v4l2_ctrl_modify_range` — Unlocked variant of `v4l2_ctrl_modify_range`

## Synopsis

```
int __v4l2_ctrl_modify_range (struct v4l2_ctrl * ctrl, s64 min, s64 max,  
u64 step, s64 def);
```

## Arguments

*ctrl*    The control to update.

*min*    The control's minimum value.

*max*    The control's maximum value.

*step*   The control's step value

*def*    The control's default value.

## Description

Update the range of a control on the fly. This works for control types INTEGER, BOOLEAN, MENU, INTEGER MENU and BITMASK. For menu controls the *step* value is interpreted as a `menu_skip_mask`.

An error is returned if one of the range arguments is invalid for this control type.

This function assumes that the control handler is not locked and will take the lock itself.

## Name

`v4l2_ctrl_modify_range` — Update the range of a control.

## Synopsis

```
int v4l2_ctrl_modify_range (struct v4l2_ctrl * ctrl, s64 min, s64 max,  
u64 step, s64 def);
```

## Arguments

*ctrl*    The control to update.

*min*    The control's minimum value.

*max*    The control's maximum value.

*step*   The control's step value

*def*    The control's default value.

## Description

Update the range of a control on the fly. This works for control types INTEGER, BOOLEAN, MENU, INTEGER MENU and BITMASK. For menu controls the *step* value is interpreted as a `menu_skip_mask`.

An error is returned if one of the range arguments is invalid for this control type.

This function assumes that the control handler is not locked and will take the lock itself.

## Name

`v4l2_ctrl_notify` — Function to set a notify callback for a control.

## Synopsis

```
void v4l2_ctrl_notify (struct v4l2_ctrl * ctrl, v4l2_ctrl_notify_fnc  
notify, void * priv);
```

## Arguments

*ctrl*      The control.

*notify*    The callback function.

*priv*      The callback private handle, passed as argument to the callback.

## Description

This function sets a callback function for the control. If *ctrl* is NULL, then it will do nothing. If *notify* is NULL, then the notify callback will be removed.

There can be only one notify. If another already exists, then a WARN\_ON will be issued and the function will do nothing.

## Name

`v4l2_ctrl_get_name` — Get the name of the control

## Synopsis

```
const char * v4l2_ctrl_get_name (u32 id);
```

## Arguments

*id* The control ID.

## Description

This function returns the name of the given control ID or NULL if it isn't a known control.

## Name

`v4l2_ctrl_get_menu` — Get the menu string array of the control

## Synopsis

```
const char * const * v4l2_ctrl_get_menu (u32 id);
```

## Arguments

*id* The control ID.

## Description

This function returns the NULL-terminated menu string array name of the given control ID or NULL if it isn't a known menu control.

## Name

`v4l2_ctrl_get_int_menu` — Get the integer menu array of the control

## Synopsis

```
const s64 * v4l2_ctrl_get_int_menu (u32 id, u32 * len);
```

## Arguments

*id*     The control ID.

*len*    The size of the integer array.

## Description

This function returns the integer array of the given control ID or NULL if it isn't a known integer menu control.

## Name

`v4l2_ctrl_g_ctrl` — Helper function to get the control's value from within a driver.

## Synopsis

```
s32 v4l2_ctrl_g_ctrl (struct v4l2_ctrl * ctrl);
```

## Arguments

*ctrl* The control.

## Description

This returns the control's value safely by going through the control framework. This function will lock the control's handler, so it cannot be used from within the `v4l2_ctrl_ops` functions.

This function is for integer type controls only.



## Name

`__v4l2_ctrl_s_ctrl` — Unlocked variant of `v4l2_ctrl_s_ctrl`.

## Synopsis

```
int __v4l2_ctrl_s_ctrl (struct v4l2_ctrl * ctrl, s32 val);
```

## Arguments

*ctrl*    The control.

*val*     The new value.

## Description

This set the control's new value safely by going through the control framework. This function will lock the control's handler, so it cannot be used from within the `v4l2_ctrl_ops` functions.

This function is for integer type controls only.

## Name

`v4l2_ctrl_g_ctrl_int64` — Helper function to get a 64-bit control's value from within a driver.

## Synopsis

```
s64 v4l2_ctrl_g_ctrl_int64 (struct v4l2_ctrl * ctrl);
```

## Arguments

*ctrl* The control.

## Description

This returns the control's value safely by going through the control framework. This function will lock the control's handler, so it cannot be used from within the `v4l2_ctrl_ops` functions.

This function is for 64-bit integer type controls only.

## Name

`__v4l2_ctrl_s_ctrl_int64` — Unlocked variant of `v4l2_ctrl_s_ctrl_int64`.

## Synopsis

```
int __v4l2_ctrl_s_ctrl_int64 (struct v4l2_ctrl * ctrl, s64 val);
```

## Arguments

*ctrl*    The control.

*val*     The new value.

## Description

This set the control's new value safely by going through the control framework. This function will lock the control's handler, so it cannot be used from within the `v4l2_ctrl_ops` functions.

This function is for 64-bit integer type controls only.

## Name

`v4l2_check_dv_timings_fnc` — timings check callback

## Synopsis

```
typedef bool v4l2_check_dv_timings_fnc (const struct v4l2_dv_timings *  
t, void * handle);
```

## Arguments

*t*            the `v4l2_dv_timings` struct.

*handle*    a handle from the driver.

## Description

Returns true if the given timings are valid.

## Name

`v4l2_valid_dv_timings` — are these timings valid?

## Synopsis

```
bool v4l2_valid_dv_timings (const struct v4l2_dv_timings * t, const
struct v4l2_dv_timings_cap * cap, v4l2_check_dv_timings_fnc fnc, void
* fnc_handle);
```

## Arguments

<i>t</i>	the <code>v4l2_dv_timings</code> struct.
<i>cap</i>	the <code>v4l2_dv_timings_cap</code> capabilities.
<i>fnc</i>	callback to check if this timing is OK. May be NULL.
<i>fnc_handle</i>	a handle that is passed on to <i>fnc</i> .

## Description

Returns true if the given `dv_timings` struct is supported by the hardware capabilities and the callback function (if non-NULL), returns false otherwise.

## Name

`v4l2_enum_dv_timings_cap` — Helper function to enumerate possible DV timings based on capabilities

## Synopsis

```
int v4l2_enum_dv_timings_cap (struct v4l2_enum_dv_timings * t, const
struct v4l2_dv_timings_cap * cap, v4l2_check_dv_timings_fnc fnc, void
* fnc_handle);
```

## Arguments

<i>t</i>	the <code>v4l2_enum_dv_timings</code> struct.
<i>cap</i>	the <code>v4l2_dv_timings_cap</code> capabilities.
<i>fnc</i>	callback to check if this timing is OK. May be NULL.
<i>fnc_handle</i>	a handle that is passed on to <i>fnc</i> .

## Description

This enumerates `dv_timings` using the full list of possible CEA-861 and DMT timings, filtering out any timings that are not supported based on the hardware capabilities and the callback function (if non-NULL).

If a valid timing for the given index is found, it will fill in *t* and return 0, otherwise it returns -EINVAL.

## Name

`v4l2_find_dv_timings_cap` — Find the closest timings struct

## Synopsis

```
bool    v4l2_find_dv_timings_cap    (struct    v4l2_dv_timings    *    t,
const    struct    v4l2_dv_timings_cap    *    cap,    unsigned    pclock_delta,
v4l2_check_dv_timings_fnc fnc, void * fnc_handle);
```

## Arguments

<i>t</i>	the <code>v4l2_enum_dv_timings</code> struct.
<i>cap</i>	the <code>v4l2_dv_timings_cap</code> capabilities.
<i>pclock_delta</i>	maximum delta between <code>t-&gt;pixelclock</code> and the timing struct under consideration.
<i>fnc</i>	callback to check if a given timings struct is OK. May be NULL.
<i>fnc_handle</i>	a handle that is passed on to <i>fnc</i> .

## Description

This function tries to map the given timings to an entry in the full list of possible CEA-861 and DMT timings, filtering out any timings that are not supported based on the hardware capabilities and the callback function (if non-NULL).

On success it will fill in *t* with the found timings and it returns true. On failure it will return false.

## Name

`v4l2_match_dv_timings` — do two timings match?

## Synopsis

```
bool v4l2_match_dv_timings (const struct v4l2_dv_timings * measured,  
const struct v4l2_dv_timings * standard, unsigned pclock_delta);
```

## Arguments

*measured*            the measured timings data.

*standard*           the timings according to the standard.

*pclock\_delta*       maximum delta in Hz between standard->pixelclock and the measured timings.

## Description

Returns true if the two timings match, returns false otherwise.



## Name

`v4l2_print_dv_timings` — log the contents of a `dv_timings` struct

## Synopsis

```
void v4l2_print_dv_timings (const char * dev_prefix, const char * prefix,  
const struct v4l2_dv_timings * t, bool detailed);
```

## Arguments

<i>dev_prefix</i>	device prefix for each log line.
<i>prefix</i>	additional prefix for each log line, may be NULL.
<i>t</i>	the timings data.
<i>detailed</i>	if true, give a detailed log.

## Name

`v4l2_detect_cvt` — detect if the given timings follow the CVT standard

## Synopsis

```
bool v4l2_detect_cvt (unsigned frame_height, unsigned hfreq, unsigned
vsync, unsigned active_width, u32 polarities, bool interlaced, struct
v4l2_dv_timings * fmt);
```

## Arguments

<i>frame_height</i>	the total height of the frame (including blanking) in lines.
<i>hfreq</i>	the horizontal frequency in Hz.
<i>vsync</i>	the height of the vertical sync in lines.
<i>active_width</i>	active width of image (does not include blanking). This information is needed only in case of version 2 of reduced blanking. In other cases, this parameter does not have any effect on timings.
<i>polarities</i>	the horizontal and vertical polarities (same as struct <code>v4l2_bt_timings</code> polarities).
<i>interlaced</i>	if this flag is true, it indicates interlaced format
<i>fmt</i>	the resulting timings.

## Description

This function will attempt to detect if the given values correspond to a valid CVT format. If so, then it will return true, and `fmt` will be filled in with the found CVT timings.

## Name

`v4l2_detect_gtf` — detect if the given timings follow the GTF standard

## Synopsis

```
bool v4l2_detect_gtf (unsigned frame_height, unsigned hfreq, unsigned
vsync, u32 polarities, bool interlaced, struct v4l2_fract aspect, struct
v4l2_dv_timings * fmt);
```

## Arguments

<i>frame_height</i>	the total height of the frame (including blanking) in lines.
<i>hfreq</i>	the horizontal frequency in Hz.
<i>vsync</i>	the height of the vertical sync in lines.
<i>polarities</i>	the horizontal and vertical polarities (same as struct v4l2_bt_timings polarities).
<i>interlaced</i>	if this flag is true, it indicates interlaced format
<i>aspect</i>	preferred aspect ratio. GTF has no method of determining the aspect ratio in order to derive the image width from the image height, so it has to be passed explicitly. Usually the native screen aspect ratio is used for this. If it is not filled in correctly, then 16:9 will be assumed.
<i>fmt</i>	the resulting timings.

## Description

This function will attempt to detect if the given values correspond to a valid GTF format. If so, then it will return true, and `fmt` will be filled in with the found GTF timings.

## Name

`v4l2_calc_aspect_ratio` — calculate the aspect ratio based on bytes 0x15 and 0x16 from the EDID.

## Synopsis

```
struct v4l2_fract  v4l2_calc_aspect_ratio (u8  hor_landscape,  u8
vert_portrait);
```

## Arguments

*hor\_landscape* byte 0x15 from the EDID.

*vert\_portrait* byte 0x16 from the EDID.

## Description

Determines the aspect ratio from the EDID. See VESA Enhanced EDID standard, release A, rev 2, section 3.6.2: “Horizontal and Vertical Screen Size or Aspect Ratio”

## Name

struct v4l2\_kevent — Internal kernel event struct.

## Synopsis

```
struct v4l2_kevent {  
    struct list_head list;  
    struct v4l2_subscribed_event * sev;  
    struct v4l2_event event;  
};
```

## Members

list	List node for the v4l2_fh->available list.
sev	Pointer to parent v4l2_subscribed_event.
event	The event itself.

## Name

struct v4l2\_subscribed\_event — Internal struct representing a subscribed event.

## Synopsis

```
struct v4l2_subscribed_event {
    struct list_head list;
    u32 type;
    u32 id;
    u32 flags;
    struct v4l2_fh * fh;
    struct list_head node;
    const struct v4l2_subscribed_event_ops * ops;
    unsigned elems;
    unsigned first;
    unsigned in_use;
    struct v4l2_kevent events[];
};
```

## Members

list	List node for the v4l2_fh->subscribed list.
type	Event type.
id	Associated object ID (e.g. control ID). 0 if there isn't any.
flags	Copy of v4l2_event_subscription->flags.
fh	Filehandle that subscribed to this event.
node	List node that hooks into the object's event list (if there is one).
ops	v4l2_subscribed_event_ops
elems	The number of elements in the events array.
first	The index of the events containing the oldest available event.
in_use	The number of queued events.
events[]	An array of <i>elems</i> events.

## Name

struct v4l2\_flash\_config — V4L2 Flash sub-device initialization data

## Synopsis

```
struct v4l2_flash_config {
    char dev_name[32];
    struct led_flash_setting torch_intensity;
    struct led_flash_setting indicator_intensity;
    u32 flash_faults;
    unsigned int has_external_strobe:1;
};
```

## Members

dev_name[32]	the name of the media entity, unique in the system
torch_intensity	constraints for the LED in torch mode
indicator_intensity	constraints for the indicator LED
flash_faults	bitmask of flash faults that the LED flash class device can report; corresponding LED_FAULT* bit definitions are available in the header file <linux/led-class-flash.h>
has_external_strobe	external strobe capability

## Name

struct v4l2\_flash — Flash sub-device context

## Synopsis

```
struct v4l2_flash {  
    struct led_classdev_flash * fled_cdev;  
    struct led_classdev_flash * iled_cdev;  
    const struct v4l2_flash_ops * ops;  
    struct v4l2_subdev sd;  
    struct v4l2_ctrl_handler hdl;  
    struct v4l2_ctrl ** ctrls;  
};
```

## Members

fled_cdev	LED flash class device controlled by this sub-device
iled_cdev	LED class device representing indicator LED associated with the LED flash class device
ops	V4L2 specific flash ops
sd	V4L2 sub-device
hdl	flash controls handler
ctrls	array of pointers to controls, whose values define the sub-device state



## Name

`v4l2_flash_init` — initialize V4L2 flash led sub-device

## Synopsis

```
struct v4l2_flash * v4l2_flash_init (struct device * dev, struct
device_node * of_node, struct led_classdev_flash * fled_cdev, struct
led_classdev_flash * iled_cdev, const struct v4l2_flash_ops * ops,
struct v4l2_flash_config * config);
```

## Arguments

<i>dev</i>	flash device, e.g. an I2C device
<i>of_node</i>	<i>of_node</i> of the LED, may be NULL if the same as device's
<i>fled_cdev</i>	LED flash class device to wrap
<i>iled_cdev</i>	LED flash class device representing indicator LED associated with <i>fled_cdev</i> , may be NULL
<i>ops</i>	V4L2 Flash device ops
<i>config</i>	initialization data for V4L2 Flash sub-device

## Description

Create V4L2 Flash sub-device wrapping given LED subsystem device.

## Returns

A valid pointer, or, when an error occurs, the return value is encoded using `ERR_PTR`. Use `IS_ERR` to check and `PTR_ERR` to obtain the numeric return value.

## Name

`v4l2_flash_release` — release V4L2 Flash sub-device

## Synopsis

```
void v4l2_flash_release (struct v4l2_flash * v4l2_flash);
```

## Arguments

*v4l2\_flash* the V4L2 Flash sub-device to release

## Description

Release V4L2 Flash sub-device.

## Name

enum v4l2\_mbus\_type — media bus type

## Synopsis

```
enum v4l2_mbus_type {  
    V4L2_MBUS_PARALLEL,  
    V4L2_MBUS_BT656,  
    V4L2_MBUS_CSI2  
};
```

## Constants

V4L2\_MBUS\_PARALLEL parallel interface with hsync and vsync

V4L2\_MBUS\_BT656 parallel interface with embedded synchronisation, can also be used for BT.1120

V4L2\_MBUS\_CSI2 MIPI CSI-2 serial interface

## Name

struct v4l2\_mbus\_config — media bus configuration

## Synopsis

```
struct v4l2_mbus_config {  
    enum v4l2_mbus_type type;  
    unsigned int flags;  
};
```

## Members

type      in: interface type

flags     in / out: configuration flags, depending on *type*

## Name

struct v4l2\_m2m\_ops — mem-to-mem device driver callbacks

## Synopsis

```
struct v4l2_m2m_ops {  
    void (* device_run) (void *priv);  
    int (* job_ready) (void *priv);  
    void (* job_abort) (void *priv);  
    void (* lock) (void *priv);  
    void (* unlock) (void *priv);  
};
```

## Members

device_run	required. Begin the actual job (transaction) inside this callback. The job does NOT have to end before this callback returns (and it will be the usual case). When the job finishes, v4l2_m2m_job_finish has to be called.
job_ready	optional. Should return 0 if the driver does not have a job fully prepared to run yet (i.e. it will not be able to finish a transaction without sleeping). If not provided, it will be assumed that one source and one destination buffer are all that is required for the driver to perform one full transaction. This method may not sleep.
job_abort	required. Informs the driver that it has to abort the currently running transaction as soon as possible (i.e. as soon as it can stop the device safely; e.g. in the next interrupt handler), even if the transaction would not have been finished by then. After the driver performs the necessary steps, it has to call v4l2_m2m_job_finish (as if the transaction ended normally). This function does not have to (and will usually not) wait until the device enters a state when it can be stopped.
lock	optional. Define a driver's own lock callback, instead of using m2m_ctx->q_lock.
unlock	optional. Define a driver's own unlock callback, instead of using m2m_ctx->q_lock.

## Name

`v4l2_m2m_num_src_bufs_ready` — return the number of source buffers ready for use

## Synopsis

```
unsigned int  v4l2_m2m_num_src_bufs_ready (struct v4l2_m2m_ctx *  
m2m_ctx);
```

## Arguments

*m2m\_ctx* pointer to struct `v4l2_m2m_ctx`

## Name

`v4l2_m2m_num_dst_bufs_ready` — return the number of destination buffers ready for use

## Synopsis

```
unsigned int  v4l2_m2m_num_dst_bufs_ready (struct v4l2_m2m_ctx *  
m2m_ctx);
```

## Arguments

*m2m\_ctx* pointer to struct `v4l2_m2m_ctx`

## Name

`v4l2_m2m_next_src_buf` — return next source buffer from the list of ready buffers

## Synopsis

```
void * v4l2_m2m_next_src_buf (struct v4l2_m2m_ctx * m2m_ctx);
```

## Arguments

*m2m\_ctx* pointer to struct `v4l2_m2m_ctx`



## Name

`v4l2_m2m_next_dst_buf` — return next destination buffer from the list of ready buffers

## Synopsis

```
void * v4l2_m2m_next_dst_buf (struct v4l2_m2m_ctx * m2m_ctx);
```

## Arguments

*m2m\_ctx* pointer to struct `v4l2_m2m_ctx`

## Name

`v4l2_m2m_get_src_vq` — return `vb2_queue` for source buffers

## Synopsis

```
struct vb2_queue * v4l2_m2m_get_src_vq (struct v4l2_m2m_ctx * m2m_ctx);
```

## Arguments

*m2m\_ctx*   pointer to struct `v4l2_m2m_ctx`

## Name

`v4l2_m2m_get_dst_vq` — return `vb2_queue` for destination buffers

## Synopsis

```
struct vb2_queue * v4l2_m2m_get_dst_vq (struct v4l2_m2m_ctx * m2m_ctx);
```

## Arguments

*m2m\_ctx* pointer to struct `v4l2_m2m_ctx`

## Name

`v4l2_m2m_src_buf_remove` — take off a source buffer from the list of ready buffers and return it

## Synopsis

```
void * v4l2_m2m_src_buf_remove (struct v4l2_m2m_ctx * m2m_ctx);
```

## Arguments

*m2m\_ctx* pointer to struct `v4l2_m2m_ctx`

## Name

`v4l2_m2m_dst_buf_remove` — take off a destination buffer from the list of ready buffers and return it

## Synopsis

```
void * v4l2_m2m_dst_buf_remove (struct v4l2_m2m_ctx * m2m_ctx);
```

## Arguments

*m2m\_ctx* pointer to struct `v4l2_m2m_ctx`

## Name

struct v4l2\_of\_bus\_mipi\_csi2 — MIPI CSI-2 bus data structure

## Synopsis

```
struct v4l2_of_bus_mipi_csi2 {  
    unsigned int flags;  
    unsigned char data_lanes[4];  
    unsigned char clock_lane;  
    unsigned short num_data_lanes;  
    bool lane_polarities[5];  
};
```

## Members

flags	media bus (V4L2_MBUS_*) flags
data_lanes[4]	an array of physical data lane indexes
clock_lane	physical lane index of the clock lane
num_data_lanes	number of data lanes
lane_polarities[5]	polarity of the lanes. The order is the same of the physical lanes.

## Name

struct v4l2\_of\_bus\_parallel — parallel data bus data structure

## Synopsis

```
struct v4l2_of_bus_parallel {  
    unsigned int flags;  
    unsigned char bus_width;  
    unsigned char data_shift;  
};
```

## Members

flags	media bus (V4L2_MBUS_*) flags
bus_width	bus width in bits
data_shift	data shift in bits

## Name

struct v4l2\_of\_endpoint — the endpoint data structure

## Synopsis

```
struct v4l2_of_endpoint {
    struct of_endpoint base;
    enum v4l2_mbus_type bus_type;
    union bus;
    u64 * link_frequencies;
    unsigned int nr_of_link_frequencies;
};
```

## Members

base	struct of_endpoint containing port, id, and local of_node
bus_type	bus type
bus	bus configuration data structure
link_frequencies	array of supported link frequencies
nr_of_link_frequencies	number of elements in link_frequenccies array



## Name

struct v4l2\_of\_link — a link between two endpoints

## Synopsis

```
struct v4l2_of_link {  
    struct device_node * local_node;  
    unsigned int local_port;  
    struct device_node * remote_node;  
    unsigned int remote_port;  
};
```

## Members

local_node	pointer to device_node of this endpoint
local_port	identifier of the port this endpoint belongs to
remote_node	pointer to device_node of the remote endpoint
remote_port	identifier of the port the remote endpoint belongs to

## Name

struct v4l2\_subdev\_core\_ops — Define core ops callbacks for subdevs

## Synopsis

```
struct v4l2_subdev_core_ops {
    int (* log_status) (struct v4l2_subdev *sd);
    int (* s_io_pin_config) (struct v4l2_subdev *sd, size_t n, struct v4l2_subdev_io_
    int (* init) (struct v4l2_subdev *sd, u32 val);
    int (* load_fw) (struct v4l2_subdev *sd);
    int (* reset) (struct v4l2_subdev *sd, u32 val);
    int (* s_gpio) (struct v4l2_subdev *sd, u32 val);
    int (* queryctrl) (struct v4l2_subdev *sd, struct v4l2_queryctrl *qc);
    int (* g_ctrl) (struct v4l2_subdev *sd, struct v4l2_control *ctrl);
    int (* s_ctrl) (struct v4l2_subdev *sd, struct v4l2_control *ctrl);
    int (* g_ext_ctrls) (struct v4l2_subdev *sd, struct v4l2_ext_controls *ctrls);
    int (* s_ext_ctrls) (struct v4l2_subdev *sd, struct v4l2_ext_controls *ctrls);
    int (* try_ext_ctrls) (struct v4l2_subdev *sd, struct v4l2_ext_controls *ctrls);
    int (* querymenu) (struct v4l2_subdev *sd, struct v4l2_querymenu *qm);
    long (* ioctl) (struct v4l2_subdev *sd, unsigned int cmd, void *arg);
#ifdef CONFIG_COMPAT
    long (* compat_ioctl32) (struct v4l2_subdev *sd, unsigned int cmd, unsigned long
#endif
#ifdef CONFIG_VIDEO_ADV_DEBUG
    int (* g_register) (struct v4l2_subdev *sd, struct v4l2_dbg_register *reg);
    int (* s_register) (struct v4l2_subdev *sd, const struct v4l2_dbg_register *reg)
#endif
    int (* s_power) (struct v4l2_subdev *sd, int on);
    int (* interrupt_service_routine) (struct v4l2_subdev *sd, u32 status, bool *hand
    int (* subscribe_event) (struct v4l2_subdev *sd, struct v4l2_fh *fh, struct v4l2_
    int (* unsubscribe_event) (struct v4l2_subdev *sd, struct v4l2_fh *fh, struct v4l2_
};
```

## Members

log_status	callback for VIDIOC_LOG_STATUS ioctl handler code.
s_io_pin_config	configure one or more chip I/O pins for chips that multiplex different internal signal pads out to IO pins. This function takes a pointer to an array of 'n' pin configuration entries, one for each pin being configured. This function could be called at times other than just subdevice initialization.
init	initialize the sensor registers to some sort of reasonable default values. Do not use for new drivers and should be removed in existing drivers.
load_fw	load firmware.
reset	generic reset command. The argument selects which subsystems to reset. Passing 0 will always reset the whole chip. Do not use for new drivers without discussing this first on the linux-media mailinglist. There should be no reason normally to reset a device.

<code>s_gpio</code>	set GPIO pins. Very simple right now, might need to be extended with a direction argument if needed.
<code>queryctrl</code>	callback for <code>VIDIOC_QUERYCTL</code> ioctl handler code.
<code>g_ctrl</code>	callback for <code>VIDIOC_G_CTRL</code> ioctl handler code.
<code>s_ctrl</code>	callback for <code>VIDIOC_S_CTRL</code> ioctl handler code.
<code>g_ext_ctrls</code>	callback for <code>VIDIOC_G_EXT_CTRL</code> ioctl handler code.
<code>s_ext_ctrls</code>	callback for <code>VIDIOC_S_EXT_CTRL</code> ioctl handler code.
<code>try_ext_ctrls</code>	callback for <code>VIDIOC_TRY_EXT_CTRL</code> ioctl handler code.
<code>querymenu</code>	callback for <code>VIDIOC_QUERYMENU</code> ioctl handler code.
<code>ioctl</code>	called at the end of <code>ioctl</code> syscall handler at the V4L2 core. used to provide support for private ioctls used on the driver.
<code>compat_ioctl32</code>	called when a 32 bits application uses a 64 bits Kernel, in order to fix data passed from/to userspace.
<code>g_register</code>	callback for <code>VIDIOC_G_REGISTER</code> ioctl handler code.
<code>s_register</code>	callback for <code>VIDIOC_S_REGISTER</code> ioctl handler code.
<code>s_power</code>	puts subdevice in power saving mode ( <code>on == 0</code> ) or normal operation mode ( <code>on == 1</code> ).
<code>interrupt_service_routine</code>	Called by the bridge chip's interrupt service handler, when an interrupt status has be raised due to this subdev, so that this subdev can handle the details. It may schedule work to be performed later. It must not sleep. *Called from an IRQ context*.
<code>subscribe_event</code>	used by the drivers to request the control framework that for it to be warned when the value of a control changes.
<code>unsubscribe_event</code>	remove event subscription from the control framework.

## Name

struct v4l2\_subdev\_tuner\_ops — Callbacks used when v4l device was opened in radio mode.

## Synopsis

```
struct v4l2_subdev_tuner_ops {
    int (* s_radio) (struct v4l2_subdev *sd);
    int (* s_frequency) (struct v4l2_subdev *sd, const struct v4l2_frequency *freq);
    int (* g_frequency) (struct v4l2_subdev *sd, struct v4l2_frequency *freq);
    int (* enum_freq_bands) (struct v4l2_subdev *sd, struct v4l2_frequency_band *band);
    int (* g_tuner) (struct v4l2_subdev *sd, struct v4l2_tuner *vt);
    int (* s_tuner) (struct v4l2_subdev *sd, const struct v4l2_tuner *vt);
    int (* g_modulator) (struct v4l2_subdev *sd, struct v4l2_modulator *vm);
    int (* s_modulator) (struct v4l2_subdev *sd, const struct v4l2_modulator *vm);
    int (* s_type_addr) (struct v4l2_subdev *sd, struct tuner_setup *type);
    int (* s_config) (struct v4l2_subdev *sd, const struct v4l2_priv_tun_config *conf);
};
```

## Members

s_radio	callback for VIDIOC_S_RADIO ioctl handler code.
s_frequency	callback for VIDIOC_S_FREQUENCY ioctl handler code.
g_frequency	callback for VIDIOC_G_FREQUENCY ioctl handler code. freq->type must be filled in. Normally done by video_ioctl2 or the bridge driver.
enum_freq_bands	callback for VIDIOC_ENUM_FREQ_BANDS ioctl handler code.
g_tuner	callback for VIDIOC_G_TUNER ioctl handler code.
s_tuner	callback for VIDIOC_S_TUNER ioctl handler code. vt->type must be filled in. Normally done by video_ioctl2 or the bridge driver.
g_modulator	callback for VIDIOC_G_MODULATOR ioctl handler code.
s_modulator	callback for VIDIOC_S_MODULATOR ioctl handler code.
s_type_addr	sets tuner type and its I2C addr.
s_config	sets tda9887 specific stuff, like port1, port2 and qss

## Name

struct v4l2\_subdev\_audio\_ops — Callbacks used for audio-related settings

## Synopsis

```
struct v4l2_subdev_audio_ops {
    int (* s_clock_freq) (struct v4l2_subdev *sd, u32 freq);
    int (* s_i2s_clock_freq) (struct v4l2_subdev *sd, u32 freq);
    int (* s_routing) (struct v4l2_subdev *sd, u32 input, u32 output, u32 config);
    int (* s_stream) (struct v4l2_subdev *sd, int enable);
};
```

## Members

s_clock_freq	set the frequency (in Hz) of the audio clock output. Used to slave an audio processor to the video decoder, ensuring that audio and video remain synchronized. Usual values for the frequency are 48000, 44100 or 32000 Hz. If the frequency is not supported, then -EINVAL is returned.
s_i2s_clock_freq	sets I2S speed in bps. This is used to provide a standard way to select I2S clock used by driving digital audio streams at some board designs. Usual values for the frequency are 1024000 and 2048000. If the frequency is not supported, then -EINVAL is returned.
s_routing	used to define the input and/or output pins of an audio chip, and any additional configuration data. Never attempt to use user-level input IDs (e.g. Composite, S-Video, Tuner) at this level. An i2c device shouldn't know about whether an input pin is connected to a Composite connector, become on another board or platform it might be connected to something else entirely. The calling driver is responsible for mapping a user-level input to the right pins on the i2c device.
s_stream	used to notify the audio code that stream will start or has stopped.

## Name

struct v4l2\_mbus\_frame\_desc\_entry — media bus frame description structure

## Synopsis

```
struct v4l2_mbus_frame_desc_entry {  
    u16 flags;  
    u32 pixelcode;  
    u32 length;  
};
```

## Members

flags	V4L2_MBUS_FRAME_DESC_FL_* flags
pixelcode	media bus pixel code, valid if FRAME_DESC_FL_BLOB is not set
length	number of octets per frame, valid if V4L2_MBUS_FRAME_DESC_FL_BLOB is set

## Name

struct v4l2\_mbus\_frame\_desc — media bus data frame description

## Synopsis

```
struct v4l2_mbus_frame_desc {  
    struct v4l2_mbus_frame_desc_entry entry[V4L2_FRAME_DESC_ENTRY_MAX];  
    unsigned short num_entries;  
};
```

## Members

entry[V4L2\_FRAME\_DESC\_ENTRY\_MAX] — descriptors array

num\_entries — number of entries in *entry* array

## Name

struct v4l2\_subdev\_video\_ops — Callbacks used when v4l device was opened in video mode.

## Synopsis

```
struct v4l2_subdev_video_ops {
    int (* s_routing) (struct v4l2_subdev *sd, u32 input, u32 output, u32 config);
    int (* s_crystal_freq) (struct v4l2_subdev *sd, u32 freq, u32 flags);
    int (* g_std) (struct v4l2_subdev *sd, v4l2_std_id *norm);
    int (* s_std) (struct v4l2_subdev *sd, v4l2_std_id norm);
    int (* s_std_output) (struct v4l2_subdev *sd, v4l2_std_id std);
    int (* g_std_output) (struct v4l2_subdev *sd, v4l2_std_id *std);
    int (* querystd) (struct v4l2_subdev *sd, v4l2_std_id *std);
    int (* g_tvnorms) (struct v4l2_subdev *sd, v4l2_std_id *std);
    int (* g_tvnorms_output) (struct v4l2_subdev *sd, v4l2_std_id *std);
    int (* g_input_status) (struct v4l2_subdev *sd, u32 *status);
    int (* s_stream) (struct v4l2_subdev *sd, int enable);
    int (* cropcap) (struct v4l2_subdev *sd, struct v4l2_cropcap *cc);
    int (* g_crop) (struct v4l2_subdev *sd, struct v4l2_crop *crop);
    int (* s_crop) (struct v4l2_subdev *sd, const struct v4l2_crop *crop);
    int (* g_parm) (struct v4l2_subdev *sd, struct v4l2_streamparm *param);
    int (* s_parm) (struct v4l2_subdev *sd, struct v4l2_streamparm *param);
    int (* g_frame_interval) (struct v4l2_subdev *sd, struct v4l2_subdev_frame_interv
    int (* s_frame_interval) (struct v4l2_subdev *sd, struct v4l2_subdev_frame_interv
    int (* s_dv_timings) (struct v4l2_subdev *sd, struct v4l2_dv_timings *timings);
    int (* g_dv_timings) (struct v4l2_subdev *sd, struct v4l2_dv_timings *timings);
    int (* query_dv_timings) (struct v4l2_subdev *sd, struct v4l2_dv_timings *timings
    int (* g_mbus_config) (struct v4l2_subdev *sd, struct v4l2_mbus_config *cfg);
    int (* s_mbus_config) (struct v4l2_subdev *sd, const struct v4l2_mbus_config *cfg
    int (* s_rx_buffer) (struct v4l2_subdev *sd, void *buf, unsigned int *size);
};
```

## Members

s_routing	see s_routing in audio_ops, except this version is for video devices.
s_crystal_freq	sets the frequency of the crystal used to generate the clocks in Hz. An extra flags field allows device specific configuration regarding clock frequency dividers, etc. If not used, then set flags to 0. If the frequency is not supported, then -EINVAL is returned.
g_std	callback for VIDIOC_G_STD ioctl handler code.
s_std	callback for VIDIOC_S_STD ioctl handler code.
s_std_output	set v4l2_std_id for video OUTPUT devices. This is ignored by video input devices.
g_std_output	get current standard for video OUTPUT devices. This is ignored by video input devices.
querystd	callback for VIDIOC_QUERYSTD ioctl handler code.



<code>g_tvnorms</code>	get <code>v4l2_std_id</code> with all standards supported by the video CAPTURE device. This is ignored by video output devices.
<code>g_tvnorms_output</code>	get <code>v4l2_std_id</code> with all standards supported by the video OUTPUT device. This is ignored by video capture devices.
<code>g_input_status</code>	get input status. Same as the status field in the <code>v4l2_input</code> struct.
<code>s_stream</code>	used to notify the driver that a video stream will start or has stopped.
<code>cropcap</code>	callback for <code>VIDIOC_CROPCAP</code> ioctl handler code.
<code>g_crop</code>	callback for <code>VIDIOC_G_CROP</code> ioctl handler code.
<code>s_crop</code>	callback for <code>VIDIOC_S_CROP</code> ioctl handler code.
<code>g_parm</code>	callback for <code>VIDIOC_G_PARM</code> ioctl handler code.
<code>s_parm</code>	callback for <code>VIDIOC_S_PARM</code> ioctl handler code.
<code>g_frame_interval</code>	callback for <code>VIDIOC_G_FRAMEINTERVAL</code> ioctl handler code.
<code>s_frame_interval</code>	callback for <code>VIDIOC_S_FRAMEINTERVAL</code> ioctl handler code.
<code>s_dv_timings</code>	Set custom dv timings in the sub device. This is used when sub device is capable of setting detailed timing information in the hardware to generate/detect the video signal.
<code>g_dv_timings</code>	Get custom dv timings in the sub device.
<code>query_dv_timings</code>	callback for <code>VIDIOC_QUERY_DV_TIMINGS</code> ioctl handler code.
<code>g_mbus_config</code>	get supported mediabus configurations
<code>s_mbus_config</code>	set a certain mediabus configuration. This operation is added for compatibility with soc-camera drivers and should not be used by new software.
<code>s_rx_buffer</code>	set a host allocated memory buffer for the subdev. The subdev can adjust <i>size</i> to a lower value and must not write more data to the buffer starting at <i>data</i> than the original value of <i>size</i> .

## Name

struct v4l2\_subdev\_vbi\_ops — Callbacks used when v4l device was opened in video mode via the vbi device node.

## Synopsis

```
struct v4l2_subdev_vbi_ops {
    int (* decode_vbi_line) (struct v4l2_subdev *sd, struct v4l2_decode_vbi_line *vb
    int (* s_vbi_data) (struct v4l2_subdev *sd, const struct v4l2_sliced_vbi_data *v
    int (* g_vbi_data) (struct v4l2_subdev *sd, struct v4l2_sliced_vbi_data *vbi_dat
    int (* g_sliced_vbi_cap) (struct v4l2_subdev *sd, struct v4l2_sliced_vbi_cap *ca
    int (* s_raw_fmt) (struct v4l2_subdev *sd, struct v4l2_vbi_format *fmt);
    int (* g_sliced_fmt) (struct v4l2_subdev *sd, struct v4l2_sliced_vbi_format *fmt
    int (* s_sliced_fmt) (struct v4l2_subdev *sd, struct v4l2_sliced_vbi_format *fmt
};
```

## Members

decode_vbi_line	video decoders that support sliced VBI need to implement this ioctl. Field p of the v4l2_sliced_vbi_line struct is set to the start of the VBI data that was generated by the decoder. The driver then parses the sliced VBI data and sets the other fields in the struct accordingly. The pointer p is updated to point to the start of the payload which can be copied verbatim into the data field of the v4l2_sliced_vbi_data struct. If no valid VBI data was found, then the type field is set to 0 on return.
s_vbi_data	used to generate VBI signals on a video signal. v4l2_sliced_vbi_data is filled with the data packets that should be output. Note that if you set the line field to 0, then that VBI signal is disabled. If no valid VBI data was found, then the type field is set to 0 on return.
g_vbi_data	used to obtain the sliced VBI packet from a readback register. Not all video decoders support this. If no data is available because the readback register contains invalid or erroneous data -EIO is returned. Note that you must fill in the 'id' member and the 'field' member (to determine whether CC data from the first or second field should be obtained).
g_sliced_vbi_cap	callback for VIDIOC_SLICED_VBI_CAP ioctl handler code.
s_raw_fmt	setup the video encoder/decoder for raw VBI.
g_sliced_fmt	retrieve the current sliced VBI settings.
s_sliced_fmt	setup the sliced VBI settings.

## Name

struct v4l2\_subdev\_sensor\_ops — v4l2-subdev sensor operations

## Synopsis

```
struct v4l2_subdev_sensor_ops {  
    int (* g_skip_top_lines) (struct v4l2_subdev *sd, u32 *lines);  
    int (* g_skip_frames) (struct v4l2_subdev *sd, u32 *frames);  
};
```

## Members

<code>g_skip_top_lines</code>	number of lines at the top of the image to be skipped. This is needed for some sensors, which always corrupt several top lines of the output image, or which send their metadata in them.
<code>g_skip_frames</code>	number of frames to skip at stream start. This is needed for buggy sensors that generate faulty frames when they are turned on.

## Name

struct v4l2\_subdev\_pad\_ops — v4l2-subdev pad level operations

## Synopsis

```
struct v4l2_subdev_pad_ops {
    int (* enum_mbus_code) (struct v4l2_subdev *sd, struct v4l2_subdev_pad_config *cfg);
    int (* enum_frame_size) (struct v4l2_subdev *sd, struct v4l2_subdev_pad_config *cfg);
    int (* enum_frame_interval) (struct v4l2_subdev *sd, struct v4l2_subdev_pad_config *cfg);
    int (* get_fmt) (struct v4l2_subdev *sd, struct v4l2_subdev_pad_config *cfg, struct v4l2_subdev_format *fmt);
    int (* set_fmt) (struct v4l2_subdev *sd, struct v4l2_subdev_pad_config *cfg, struct v4l2_subdev_format *fmt);
    int (* get_selection) (struct v4l2_subdev *sd, struct v4l2_subdev_pad_config *cfg, struct v4l2_subdev_selection *sel);
    int (* set_selection) (struct v4l2_subdev *sd, struct v4l2_subdev_pad_config *cfg, struct v4l2_subdev_selection *sel);
    int (* get_edid) (struct v4l2_subdev *sd, struct v4l2_edid *edid);
    int (* set_edid) (struct v4l2_subdev *sd, struct v4l2_edid *edid);
    int (* dv_timings_cap) (struct v4l2_subdev *sd, struct v4l2_dv_timings_cap *cap);
    int (* enum_dv_timings) (struct v4l2_subdev *sd, struct v4l2_enum_dv_timings *timings);
#ifdef CONFIG_MEDIA_CONTROLLER
    int (* link_validate) (struct v4l2_subdev *sd, struct media_link *link, struct v4l2_subdev_link_config *lcfg);
#endif
    int (* get_frame_desc) (struct v4l2_subdev *sd, unsigned int pad, struct v4l2_mbus_frame_desc *fdesc);
    int (* set_frame_desc) (struct v4l2_subdev *sd, unsigned int pad, struct v4l2_mbus_frame_desc *fdesc);
};
```

## Members

enum_mbus_code	callback for VIDIOC_SUBDEV_ENUM_MBUS_CODE ioctl handler code.
enum_frame_size	callback for VIDIOC_SUBDEV_ENUM_FRAME_SIZE ioctl handler code.
enum_frame_interval	callback for VIDIOC_SUBDEV_ENUM_FRAME_INTERVAL ioctl handler code.
get_fmt	callback for VIDIOC_SUBDEV_G_FMT ioctl handler code.
set_fmt	callback for VIDIOC_SUBDEV_S_FMT ioctl handler code.
get_selection	callback for VIDIOC_SUBDEV_G_SELECTION ioctl handler code.
set_selection	callback for VIDIOC_SUBDEV_S_SELECTION ioctl handler code.
get_edid	callback for VIDIOC_SUBDEV_G_EDID ioctl handler code.
set_edid	callback for VIDIOC_SUBDEV_S_EDID ioctl handler code.
dv_timings_cap	callback for VIDIOC_SUBDEV_DV_TIMINGS_CAP ioctl handler code.
enum_dv_timings	callback for VIDIOC_SUBDEV_ENUM_DV_TIMINGS ioctl handler code.
link_validate	used by the media controller code to check if the links that belongs to a pipeline can be used for stream.

<code>get_frame_desc</code>	get the current low level media bus frame parameters.
<code>set_frame_desc</code>	set the low level media bus frame parameters, <i>fd</i> array may be adjusted by the subdev driver to device capabilities.

## Name

struct vb2\_mem\_ops — memory handling/memory allocator operations

## Synopsis

```
struct vb2_mem_ops {
    void (* alloc) (void *alloc_ctx, unsigned long size, enum dma_data_direction dma, void *cookie);
    void (* put) (void *buf_priv);
    struct dma_buf *(* get_dmabuf) (void *buf_priv, unsigned long flags);
    void *(* get_userptr) (void *alloc_ctx, unsigned long vaddr, unsigned long size, enum dma_data_direction dma, void *cookie);
    void (* put_userptr) (void *buf_priv);
    void (* prepare) (void *buf_priv);
    void (* finish) (void *buf_priv);
    void *(* attach_dmabuf) (void *alloc_ctx, struct dma_buf *dbuf, unsigned long size, enum dma_data_direction dma, void *cookie);
    void (* detach_dmabuf) (void *buf_priv);
    int (* map_dmabuf) (void *buf_priv);
    void (* unmap_dmabuf) (void *buf_priv);
    void *(* vaddr) (void *buf_priv);
    void *(* cookie) (void *buf_priv);
    unsigned int (* num_users) (void *buf_priv);
    int (* mmap) (void *buf_priv, struct vm_area_struct *vma);
};
```

## Members

alloc	allocate video memory and, optionally, allocator private data, return NULL on failure or a pointer to allocator private, per-buffer data on success; the returned private structure will then be passed as buf_priv argument to other ops in this structure. Additional gfp_flags to use when allocating the are also passed to this operation. These flags are from the gfp_flags field of vb2_queue.
put	inform the allocator that the buffer will no longer be used; usually will result in the allocator freeing the buffer (if no other users of this buffer are present); the buf_priv argument is the allocator private per-buffer structure previously returned from the alloc callback.
get_dmabuf	acquire userspace memory for a hardware operation; used for DMABUF memory types.
get_userptr	acquire userspace memory for a hardware operation; used for USERPTR memory types; vaddr is the address passed to the videobuf layer when queuing a video buffer of USERPTR type; should return an allocator private per-buffer structure associated with the buffer on success, NULL on failure; the returned private structure will then be passed as buf_priv argument to other ops in this structure.
put_userptr	inform the allocator that a USERPTR buffer will no longer be used.
prepare	called every time the buffer is passed from userspace to the driver, useful for cache synchronisation, optional.
finish	called every time the buffer is passed back from the driver to the userspace, also optional.

<code>attach_dmabuf</code>	attach a shared struct <code>dma_buf</code> for a hardware operation; used for DMABUF memory types; <code>alloc_ctx</code> is the alloc context <code>dbuf</code> is the shared <code>dma_buf</code> ; returns NULL on failure; allocator private per-buffer structure on success; this needs to be used for further accesses to the buffer.
<code>detach_dmabuf</code>	inform the exporter of the buffer that the current DMABUF buffer is no longer used; the <code>buf_priv</code> argument is the allocator private per-buffer structure previously returned from the <code>attach_dmabuf</code> callback.
<code>map_dmabuf</code>	request for access to the dmabuf from allocator; the allocator of dmabuf is informed that this driver is going to use the dmabuf.
<code>unmap_dmabuf</code>	releases access control to the dmabuf - allocator is notified that this driver is done using the dmabuf for now.
<code>vaddr</code>	return a kernel virtual address to a given memory buffer associated with the passed private structure or NULL if no such mapping exists.
<code>cookie</code>	return allocator specific cookie for a given memory buffer associated with the passed private structure or NULL if not available.
<code>num_users</code>	return the current number of users of a memory buffer; return 1 if the videobuf layer (or actually the driver using it) is the only user.
<code>mmap</code>	setup a userspace mapping for a given memory buffer under the provided virtual memory region.

## Required ops for USERPTR types

`get_userptr`, `put_userptr`.

## Required ops for MMAP types

`alloc`, `put`, `num_users`, `mmap`. Required ops for read/write access types: `alloc`, `put`, `num_users`, `vaddr`.

## Required ops for DMABUF types

`attach_dmabuf`, `detach_dmabuf`, `map_dmabuf`, `unmap_dmabuf`.

## Name

struct vb2\_plane — plane information

## Synopsis

```
struct vb2_plane {
    void * mem_priv;
    struct dma_buf * dbuf;
    unsigned int dbuf_mapped;
    unsigned int bytesused;
    unsigned int length;
    union m;
    unsigned int data_offset;
};
```

## Members

mem_priv	private data with this plane
dbuf	dma_buf - shared buffer object
dbuf_mapped	flag to show whether dbuf is mapped or not
bytesused	number of bytes occupied by data in the plane (payload)
length	size of this plane (NOT the payload) in bytes
m	Union with memtype-specific data ( <i>offset</i> , <i>userptr</i> or <i>fd</i> ).
data_offset	offset in the plane to the start of data; usually 0, unless there is a header in front of the data Should contain enough information to be able to cover all the fields of struct v4l2_plane at videodev2.h



## Name

enum vb2\_io\_modes — queue access methods

## Synopsis

```
enum vb2_io_modes {  
    VB2_MMAP,  
    VB2_USERPTR,  
    VB2_READ,  
    VB2_WRITE,  
    VB2_DMABUF  
};
```

## Constants

VB2_MMAP	driver supports MMAP with streaming API
VB2_USERPTR	driver supports USERPTR with streaming API
VB2_READ	driver supports <code>read</code> style access
VB2_WRITE	driver supports <code>write</code> style access
VB2_DMABUF	driver supports DMABUF with streaming API

## Name

enum vb2\_buffer\_state — current video buffer state

## Synopsis

```
enum vb2_buffer_state {  
    VB2_BUF_STATE_DEQUEUED,  
    VB2_BUF_STATE_PREPARING,  
    VB2_BUF_STATE_PREPARED,  
    VB2_BUF_STATE_QUEUED,  
    VB2_BUF_STATE_REQUEUEING,  
    VB2_BUF_STATE_ACTIVE,  
    VB2_BUF_STATE_DONE,  
    VB2_BUF_STATE_ERROR  
};
```

## Constants

VB2_BUF_STATE_DEQUEUED	buffer under userspace control
VB2_BUF_STATE_PREPARING	buffer is being prepared in videobuf
VB2_BUF_STATE_PREPARED	buffer prepared in videobuf and by the driver
VB2_BUF_STATE_QUEUED	buffer queued in videobuf, but not in driver
VB2_BUF_STATE_REQUEUEING	re-queue a buffer to the driver
VB2_BUF_STATE_ACTIVE	buffer queued in driver and possibly used in a hardware operation
VB2_BUF_STATE_DONE	buffer returned from driver to videobuf, but not yet dequeued to userspace
VB2_BUF_STATE_ERROR	same as above, but the operation on the buffer has ended with an error, which will be reported to the userspace when it is dequeued

## Name

struct vb2\_buffer — represents a video buffer

## Synopsis

```
struct vb2_buffer {
    struct vb2_queue * vb2_queue;
    unsigned int index;
    unsigned int type;
    unsigned int memory;
    unsigned int num_planes;
    struct vb2_plane planes[VB2_MAX_PLANES];
};
```

## Members

vb2_queue	the queue to which this driver belongs
index	id number of the buffer
type	buffer type
memory	the method, in which the actual data is passed
num_planes	number of planes in the buffer on an internal driver queue
planes[VB2_MAX_PLANES]	private per-plane information; do not change

## Name

struct vb2\_ops — driver-specific callbacks

## Synopsis

```
struct vb2_ops {
    int (* queue_setup) (struct vb2_queue *q, const void *parg, unsigned int *num_bufs);
    void (* wait_prepare) (struct vb2_queue *q);
    void (* wait_finish) (struct vb2_queue *q);
    int (* buf_init) (struct vb2_buffer *vb);
    int (* buf_prepare) (struct vb2_buffer *vb);
    void (* buf_finish) (struct vb2_buffer *vb);
    void (* buf_cleanup) (struct vb2_buffer *vb);
    int (* start_streaming) (struct vb2_queue *q, unsigned int count);
    void (* stop_streaming) (struct vb2_queue *q);
    void (* buf_queue) (struct vb2_buffer *vb);
};
```

## Members

queue_setup	called from VIDIOC_REQBUFS and VIDIOC_CREATE_BUFS handlers before memory allocation, or, if *num_planes != 0, after the allocation to verify a smaller number of buffers. Driver should return the required number of buffers in *num_buffers, the required number of planes per buffer in *num_planes; the size of each plane should be set in the sizes[] array and optional per-plane allocator specific context in the alloc_ctxs[] array. When called from VIDIOC_REQBUFS, fmt == NULL, the driver has to use the currently configured format and *num_buffers is the total number of buffers, that are being allocated. When called from VIDIOC_CREATE_BUFS, fmt != NULL and it describes the target frame format (if the format isn't valid the callback must return -EINVAL). In this case *num_buffers are being allocated additionally to q->num_buffers.
wait_prepare	release any locks taken while calling vb2 functions; it is called before an ioctl needs to wait for a new buffer to arrive; required to avoid a deadlock in blocking access type.
wait_finish	reacquire all locks released in the previous callback; required to continue operation after sleeping while waiting for a new buffer to arrive.
buf_init	called once after allocating a buffer (in MMAP case) or after acquiring a new USERPTR buffer; drivers may perform additional buffer-related initialization; initialization failure (return != 0) will prevent queue setup from completing successfully; optional.
buf_prepare	called every time the buffer is queued from userspace and from the VIDIOC_PREPARE_BUF ioctl; drivers may perform any initialization required before each hardware operation in this callback; drivers can access/modify the buffer here as it is still synced for the CPU; drivers that support VIDIOC_CREATE_BUFS must also validate the buffer size; if an error is returned, the buffer will not be queued in driver; optional.
buf_finish	called before every dequeue of the buffer back to userspace; the buffer is synced for the CPU, so drivers can access/modify the buffer contents; drivers may

	perform any operations required before userspace accesses the buffer; optional. The buffer state can be
<code>buf_cleanup</code>	called once before the buffer is freed; drivers may perform any additional cleanup; optional.
<code>start_streaming</code>	called once to enter 'streaming' state; the driver may receive buffers with <i>buf_queue</i> callback before <i>start_streaming</i> is called; the driver gets the number of already queued buffers in <i>count</i> parameter; driver can return an error if hardware fails, in that case all buffers that have been already given by the <i>buf_queue</i> callback are to be returned by the driver by calling <i>vb2_buffer_done</i> (VB2_BUF_STATE_QUEUED). If you need a minimum number of buffers before you can start streaming, then set <i>min_buffers_needed</i> in the <i>vb2_queue</i> structure. If that is non-zero then <i>start_streaming</i> won't be called until at least that many buffers have been queued up by userspace.
<code>stop_streaming</code>	called when 'streaming' state must be disabled; driver should stop any DMA transactions or wait until they finish and give back all buffers it got from <i>buf_queue</i> callback by calling <i>vb2_buffer_done</i> () with either VB2_BUF_STATE_DONE or VB2_BUF_STATE_ERROR; may use <i>vb2_wait_for_all_buffers</i> function
<code>buf_queue</code>	passes buffer <i>vb</i> to the driver; driver may start hardware operation on this buffer; driver should give the buffer back by calling <i>vb2_buffer_done</i> function; it is always called after calling <i>STREAMON</i> ioctl; might be called before <i>start_streaming</i> callback if user pre-queued buffers before calling <i>STREAMON</i> .

## one of the following

DONE and ERROR occur while streaming is in progress, and the PREPARED state occurs when the queue has been canceled and all pending buffers are being returned to their default DEQUEUED state. Typically you only have to do something if the state is VB2\_BUF\_STATE\_DONE, since in all other cases the buffer contents will be ignored anyway.

## Name

struct vb2\_queue — a videobuf queue

## Synopsis

```
struct vb2_queue {
    unsigned int type;
    unsigned int io_modes;
    unsigned fileio_read_once:1;
    unsigned fileio_write_immediately:1;
    unsigned allow_zero_bytesused:1;
    struct mutex * lock;
    void * owner;
    const struct vb2_ops * ops;
    const struct vb2_mem_ops * mem_ops;
    const struct vb2_buf_ops * buf_ops;
    void * drv_priv;
    unsigned int buf_struct_size;
    u32 timestamp_flags;
    gfp_t gfp_flags;
    u32 min_buffers_needed;
};
```

## Members

type	private buffer type whose content is defined by the vb2-core caller. For example, for V4L2, it should match the V4L2_BUF_TYPE_* in include/uapi/linux/videodev2.h
io_modes	supported io methods (see vb2_io_modes enum)
fileio_read_once	report EOF after reading the first buffer
fileio_write_immediately	queue buffer after each <code>write</code> call
allow_zero_bytesused	allow <code>bytesused == 0</code> to be passed to the driver
lock	pointer to a mutex that protects the vb2_queue struct. The driver can set this to a mutex to let the v4l2 core serialize the queuing ioctls. If the driver wants to handle locking itself, then this should be set to NULL. This lock is not used by the videobuf2 core API.
owner	The filehandle that 'owns' the buffers, i.e. the filehandle that called <code>reqbufs</code> , <code>create_buffers</code> or started <code>fileio</code> . This field is not used by the videobuf2 core API, but it allows drivers to easily associate an owner filehandle with the queue.
ops	driver-specific callbacks
mem_ops	memory allocator specific callbacks
buf_ops	callbacks to deliver buffer information between user-space and kernel-space

<code>drv_priv</code>	driver private data
<code>buf_struct_size</code>	size of the driver-specific buffer structure; “0” indicates the driver doesn't want to use a custom buffer structure type. for example, <code>sizeof(struct vb2_v4l2_buffer)</code> will be used for v4l2.
<code>timestamp_flags</code>	Timestamp flags; <code>V4L2_BUF_FLAG_TIMESTAMP_*</code> and <code>V4L2_BUF_FLAG_TSTAMP_SRC_*</code>
<code>gfp_flags</code>	additional gfp flags used when allocating the buffers. Typically this is 0, but it may be e.g. <code>GFP_DMA</code> or <code>__GFP_DMA32</code> to force the buffer allocation to a specific memory zone.
<code>min_buffers_needed</code>	the minimum number of buffers needed before <code>start_streaming</code> can be called. Used when a DMA engine cannot be started unless at least this number of buffers have been queued into the driver.

## Name

`vb2_is_streaming` — return streaming status of the queue

## Synopsis

```
bool vb2_is_streaming (struct vb2_queue * q);
```

## Arguments

*q* videobuf queue



## Name

`vb2_fileio_is_active` — return true if fileio is active.

## Synopsis

```
bool vb2_fileio_is_active (struct vb2_queue * q);
```

## Arguments

`q` videobuf queue

## Description

This returns true if `read` or `write` is used to stream the data as opposed to stream I/O. This is almost never an important distinction, except in rare cases. One such case is that using `read` or `write` to stream a format using `V4L2_FIELD_ALTERNATE` is not allowed since there is no way you can pass the field information of each buffer to/from userspace. A driver that supports this field format should check for this in the `queue_setup` op and reject it if this function returns true.

## Name

`vb2_is_busy` — return busy status of the queue

## Synopsis

```
bool vb2_is_busy (struct vb2_queue * q);
```

## Arguments

*q* videobuf queue

## Description

This function checks if queue has any buffers allocated.

## Name

`vb2_get_drv_priv` — return driver private data associated with the queue

## Synopsis

```
void * vb2_get_drv_priv (struct vb2_queue * q);
```

## Arguments

*q* videobuf queue

## Name

`vb2_set_plane_payload` — set bytes used for the plane `plane_no`

## Synopsis

```
void vb2_set_plane_payload (struct vb2_buffer * vb, unsigned int  
plane_no, unsigned long size);
```

## Arguments

<i>vb</i>	buffer for which plane payload should be set
<i>plane_no</i>	plane number for which payload should be set
<i>size</i>	payload in bytes

## Name

`vb2_get_plane_payload` — get bytes used for the plane `plane_no`

## Synopsis

```
unsigned long vb2_get_plane_payload (struct vb2_buffer * vb, unsigned  
int plane_no);
```

## Arguments

*vb*            buffer for which plane payload should be set

*plane\_no*    plane number for which payload should be set

## Name

`vb2_plane_size` — return plane size in bytes

## Synopsis

```
unsigned long vb2_plane_size (struct vb2_buffer * vb, unsigned int  
plane_no);
```

## Arguments

*vb*            buffer for which plane size should be returned

*plane\_no*    plane number for which size should be returned

## Name

`vb2_start_streaming_called` — return streaming status of driver

## Synopsis

```
bool vb2_start_streaming_called (struct vb2_queue * q);
```

## Arguments

*q* videobuf queue

## Name

`vb2_clear_last_buffer_dequeued` — clear last buffer dequeued flag of queue

## Synopsis

```
void vb2_clear_last_buffer_dequeued (struct vb2_queue * q);
```

## Arguments

*q* videobuf queue



## Name

struct vb2\_v4l2\_buffer — video buffer information for v4l2

## Synopsis

```
struct vb2_v4l2_buffer {
    struct vb2_buffer vb2_buf;
    __u32 flags;
    __u32 field;
    struct timeval timestamp;
    struct v4l2_timecode timecode;
    __u32 sequence;
};
```

## Members

vb2_buf	video buffer 2
flags	buffer informational flags
field	enum v4l2_field; field order of the image in the buffer
timestamp	frame timestamp
timecode	frame timecode
sequence	sequence count of this frame Should contain enough information to be able to cover all the fields of struct v4l2_buffer at videodev2.h

## Name

`vb2_thread_start` — start a thread for the given queue.

## Synopsis

```
int vb2_thread_start (struct vb2_queue * q, vb2_thread_fnc fnc, void *  
priv, const char * thread_name);
```

## Arguments

<i>q</i>	videobuf queue
<i>fnc</i>	callback function
<i>priv</i>	priv pointer passed to the callback function
<i>thread_name</i>	the name of the thread. This will be prefixed with “vb2-”.

## Description

This starts a thread that will queue and dequeue until an error occurs or `vb2_thread_stop` is called.

This function should not be used for anything else but the videobuf2-dvb support. If you think you have another good use-case for this, then please contact the linux-media mailinglist first.

## Name

`vb2_thread_stop` — stop the thread for the given queue.

## Synopsis

```
int vb2_thread_stop (struct vb2_queue * q);
```

## Arguments

*q* videobuf queue

## Name

struct vb2\_vmarea\_handler — common vma refcount tracking handler

## Synopsis

```
struct vb2_vmarea_handler {  
    atomic_t * refcount;  
    void (* put) (void *arg);  
    void * arg;  
};
```

## Members

refcount	pointer to refcount entry in the buffer
put	callback to function that decreases buffer refcount
arg	argument for <i>put</i> callback

## Digital TV (DVB) devices

## Name

struct dvb\_ca\_en50221 — Structure describing a CA interface

## Synopsis

```
struct dvb_ca_en50221 {
    struct module * owner;
    int (* read_attribute_mem) (struct dvb_ca_en50221 *ca,int slot, int address);
    int (* write_attribute_mem) (struct dvb_ca_en50221 *ca,int slot, int address, u8 va);
    int (* read_cam_control) (struct dvb_ca_en50221 *ca,int slot, u8 address);
    int (* write_cam_control) (struct dvb_ca_en50221 *ca,int slot, u8 address, u8 va);
    int (* slot_reset) (struct dvb_ca_en50221 *ca, int slot);
    int (* slot_shutdown) (struct dvb_ca_en50221 *ca, int slot);
    int (* slot_ts_enable) (struct dvb_ca_en50221 *ca, int slot);
    int (* poll_slot_status) (struct dvb_ca_en50221 *ca, int slot, int open);
    void * data;
    void * private;
};
```

## Members

owner	the module owning this structure
read_attribute_mem	function for reading attribute memory on the CAM
write_attribute_mem	function for writing attribute memory on the CAM
read_cam_control	function for reading the control interface on the CAM
write_cam_control	function for reading the control interface on the CAM
slot_reset	function to reset the CAM slot
slot_shutdown	function to shutdown a CAM slot
slot_ts_enable	function to enable the Transport Stream on a CAM slot
poll_slot_status	function to poll slot status. Only necessary if DVB_CA_FLAG_EN50221_IRQ_CAMCHANGE is not set.
data	private data, used by caller.
private	Opaque data used by the dvb_ca core. Do not modify!

## NOTE

the read\_\*, write\_\* and poll\_slot\_status functions will be called for different slots concurrently and need to use locks where and if appropriate. There will be no concurrent access to one slot.

## Name

`dvb_ca_en50221_camchange_irq` — A CAMCHANGE IRQ has occurred.

## Synopsis

```
void dvb_ca_en50221_camchange_irq (struct dvb_ca_en50221 * pubca, int  
slot, int change_type);
```

## Arguments

<i>pubca</i>	CA instance.
<i>slot</i>	Slot concerned.
<i>change_type</i>	One of the DVB_CA_CAMCHANGE_* values

## Name

`dvb_ca_en50221_camready_irq` — A CAMREADY IRQ has occurred.

## Synopsis

```
void dvb_ca_en50221_camready_irq (struct dvb_ca_en50221 * pubca, int  
slot);
```

## Arguments

*pubca* CA instance.

*slot* Slot concerned.

## Name

`dvb_ca_en50221_frda_irq` — An FR or a DA IRQ has occurred.

## Synopsis

```
void dvb_ca_en50221_frda_irq (struct dvb_ca_en50221 * ca, int slot);
```

## Arguments

*ca*     CA instance.

*slot*   Slot concerned.



## Name

`dvb_ca_en50221_init` — Initialise a new DVB CA device.

## Synopsis

```
int dvb_ca_en50221_init (struct dvb_adapter * dvb_adapter, struct
dvb_ca_en50221 * ca, int flags, int slot_count);
```

## Arguments

*dvb\_adapter*    DVB adapter to attach the new CA device to.

*ca*             The dvb\_ca instance.

*flags*           Flags describing the CA device (DVB\_CA\_EN50221\_FLAG\_\*).

*slot\_count*    Number of slots supported.

## Description

*return* 0 on success, nonzero on failure

## Name

`dvb_ca_en50221_release` — Release a DVB CA device.

## Synopsis

```
void dvb_ca_en50221_release (struct dvb_ca_en50221 * ca);
```

## Arguments

*ca* The associated `dvb_ca` instance.

## Name

struct dvb\_frontend\_tune\_settings — parameters to adjust frontend tuning

## Synopsis

```
struct dvb_frontend_tune_settings {  
    int min_delay_ms;  
    int step_size;  
    int max_drift;  
};
```

## Members

min_delay_ms	minimum delay for tuning, in ms
step_size	step size between two consecutive frequencies
max_drift	maximum drift

## NOTE

step\_size is in Hz, for terrestrial/cable or kHz for satellite

## Name

struct dvb\_tuner\_info — Frontend name and min/max ranges/bandwidths

## Synopsis

```
struct dvb_tuner_info {  
    char name[128];  
    u32 frequency_min;  
    u32 frequency_max;  
    u32 frequency_step;  
    u32 bandwidth_min;  
    u32 bandwidth_max;  
    u32 bandwidth_step;  
};
```

## Members

name[128]	name of the Frontend
frequency_min	minimal frequency supported
frequency_max	maximum frequency supported
frequency_step	frequency step
bandwidth_min	minimal frontend bandwidth supported
bandwidth_max	maximum frontend bandwidth supported
bandwidth_step	frontend bandwidth step

## NOTE

frequency parameters are in Hz, for terrestrial/cable or kHz for satellite.

## Name

struct analog\_parameters — Parameters to tune into an analog/radio channel

## Synopsis

```
struct analog_parameters {
    unsigned int frequency;
    unsigned int mode;
    unsigned int audmode;
    u64 std;
};
```

## Members

frequency	Frequency used by analog TV tuner (either in 62.5 kHz step, for TV, or 62.5 Hz for radio)
mode	Tuner mode, as defined on enum v4l2_tuner_type
audmode	Audio mode as defined for the rxsubchans field at videodev2.h, e. g. V4L2_TUNER_MODE_*
std	TV standard bitmap as defined at videodev2.h, e. g. V4L2_STD_*

## Description

Hybrid tuners should be supported by both V4L2 and DVB APIs. This struct contains the data that are used by the V4L2 side. To avoid dependencies from V4L2 headers, all enums here are declared as integers.

## Name

enum dvbfe\_algo — defines the algorithm used to tune into a channel

## Synopsis

```
enum dvbfe_algo {  
    DVBFE_ALGO_HW,  
    DVBFE_ALGO_SW,  
    DVBFE_ALGO_CUSTOM,  
    DVBFE_ALGO_RECOVERY  
};
```

## Constants

DVBFE_ALGO_HW	Hardware Algorithm - Devices that support this algorithm do everything in hardware and no software support is needed to handle them. Requesting these devices to LOCK is the only thing required, device is supposed to do everything in the hardware.
DVBFE_ALGO_SW	Software Algorithm - These are dumb devices, that require software to do everything
DVBFE_ALGO_CUSTOM	Customizable Algorithm - Devices having this algorithm can be customized to have specific algorithms in the frontend driver, rather than simply doing a software zig-zag. In this case the zigzag maybe hardware assisted or it maybe completely done in hardware. In all cases, usage of this algorithm, in conjunction with the search and track callbacks, utilizes the driver specific algorithm.
DVBFE_ALGO_RECOVERY	Recovery Algorithm - These devices have AUTO recovery capabilities from LOCK failure

## Name

enum dvbfe\_search — search callback possible return status

## Synopsis

```
enum dvbfe_search {  
    DVBFE_ALGO_SEARCH_SUCCESS,  
    DVBFE_ALGO_SEARCH_ASLEEP,  
    DVBFE_ALGO_SEARCH_FAILED,  
    DVBFE_ALGO_SEARCH_INVALID,  
    DVBFE_ALGO_SEARCH_AGAIN,  
    DVBFE_ALGO_SEARCH_ERROR  
};
```

## Constants

**DVBFE\_ALGO\_SEARCH\_SUCCESS** The frontend search algorithm completed and returned successfully

**DVBFE\_ALGO\_SEARCH\_ASLEEP** The frontend search algorithm is sleeping

**DVBFE\_ALGO\_SEARCH\_FAILED** The frontend search for a signal failed

**DVBFE\_ALGO\_SEARCH\_INVALID** The frontend search algorithm was probably supplied with invalid parameters and the search is an invalid one

**DVBFE\_ALGO\_SEARCH\_AGAIN** The frontend search algorithm was requested to search again

**DVBFE\_ALGO\_SEARCH\_ERROR** The frontend search algorithm failed due to some error

## Name

struct dvb\_tuner\_ops — Tuner information and callbacks

## Synopsis

```
struct dvb_tuner_ops {
    struct dvb_tuner_info info;
    int (* release) (struct dvb_frontend *fe);
    int (* init) (struct dvb_frontend *fe);
    int (* sleep) (struct dvb_frontend *fe);
    int (* suspend) (struct dvb_frontend *fe);
    int (* resume) (struct dvb_frontend *fe);
    int (* set_params) (struct dvb_frontend *fe);
    int (* set_analog_params) (struct dvb_frontend *fe, struct analog_parameters *p);
    int (* calc_regs) (struct dvb_frontend *fe, u8 *buf, int buf_len);
    int (* set_config) (struct dvb_frontend *fe, void *priv_cfg);
    int (* get_frequency) (struct dvb_frontend *fe, u32 *frequency);
    int (* get_bandwidth) (struct dvb_frontend *fe, u32 *bandwidth);
    int (* get_if_frequency) (struct dvb_frontend *fe, u32 *frequency);
#define TUNER_STATUS_LOCKED 1
#define TUNER_STATUS_STEREO 2
    int (* get_status) (struct dvb_frontend *fe, u32 *status);
    int (* get_rf_strength) (struct dvb_frontend *fe, u16 *strength);
    int (* get_afc) (struct dvb_frontend *fe, s32 *afc);
    int (* set_frequency) (struct dvb_frontend *fe, u32 frequency);
    int (* set_bandwidth) (struct dvb_frontend *fe, u32 bandwidth);
    int (* set_state) (struct dvb_frontend *fe, enum tuner_param param, struct tuner_state *state);
    int (* get_state) (struct dvb_frontend *fe, enum tuner_param param, struct tuner_state *state);
};
```

## Members

info	embedded struct dvb_tuner_info with tuner properties
release	callback function called when frontend is detached. drivers should free any allocated memory.
init	callback function used to initialize the tuner device.
sleep	callback function used to put the tuner to sleep.
suspend	callback function used to inform that the Kernel will suspend.
resume	callback function used to inform that the Kernel is resuming from suspend.
set_params	callback function used to inform the tuner to tune into a digital TV channel. The properties to be used are stored at <i>dvb_frontend.dtv_property_cache</i> ; The tuner demod can change the parameters to reflect the changes needed for the channel to be tuned, and update statistics.
set_analog_params	callback function used to tune into an analog TV channel on hybrid tuners. It passes <i>analog_parameters</i> ; to the driver.
calc_regs	callback function used to pass register data settings for simple tuners.



<code>set_config</code>	callback function used to send some tuner-specific parameters.
<code>get_frequency</code>	get the actual tuned frequency
<code>get_bandwidth</code>	get the bandwidth used by the low pass filters
<code>get_if_frequency</code>	get the Intermediate Frequency, in Hz. For baseband, should return 0.
<code>get_status</code>	returns the frontend lock status
<code>get_rf_strength</code>	returns the RF signal strength. Used mostly to support analog TV and radio. Digital TV should report, instead, via DVBv5 API ( <i>dvb_frontend.dtv_property_cache</i> ).
<code>get_afc</code>	Used only by analog TV core. Reports the frequency drift due to AFC.
<code>set_frequency</code>	Set a new frequency. Please notice that using <code>set_params</code> is preferred.
<code>set_bandwidth</code>	Set a new frequency. Please notice that using <code>set_params</code> is preferred.
<code>set_state</code>	callback function used on some legacy drivers that don't implement <code>set_params</code> in order to set properties. Shouldn't be used on new drivers.
<code>get_state</code>	callback function used to get properties by some legacy drivers that don't implement <code>set_params</code> . Shouldn't be used on new drivers.

## NOTE

frequencies used on `get_frequency` and `set_frequency` are in Hz for terrestrial/cable or kHz for satellite.

## Name

struct analog\_demod\_info — Information struct for analog TV part of the demod

## Synopsis

```
struct analog_demod_info {  
    char * name;  
};
```

## Members

name    Name of the analog TV demodulator

## Name

struct analog\_demod\_ops — Demodulation information and callbacks for analog TV and radio

## Synopsis

```
struct analog_demod_ops {
    struct analog_demod_info info;
    void (* set_params) (struct dvb_frontend *fe, struct analog_parameters *params);
    int (* has_signal) (struct dvb_frontend *fe, u16 *signal);
    int (* get_afc) (struct dvb_frontend *fe, s32 *afc);
    void (* tuner_status) (struct dvb_frontend *fe);
    void (* standby) (struct dvb_frontend *fe);
    void (* release) (struct dvb_frontend *fe);
    int (* i2c_gate_ctrl) (struct dvb_frontend *fe, int enable);
    int (* set_config) (struct dvb_frontend *fe, void *priv_cfg);
};
```

## Members

info	pointer to struct analog_demod_info
set_params	callback function used to inform the demod to set the demodulator parameters needed to decode an analog or radio channel. The properties are passed via struct <i>analog_params</i> ;
has_signal	returns 0xffff if has signal, or 0 if it doesn't.
get_afc	Used only by analog TV core. Reports the frequency drift due to AFC.
tuner_status	callback function that returns tuner status bits, e. g. TUNER_STATUS_LOCKED and TUNER_STATUS_STEREO.
standby	set the tuner to standby mode.
release	callback function called when frontend is detached. drivers should free any allocated memory.
i2c_gate_ctrl	controls the I2C gate. Newer drivers should use I2C mux support instead.
set_config	callback function used to send some tuner-specific parameters.

## Name

struct dvb\_frontend\_ops — Demodulation information and callbacks for ditiait TV

## Synopsis

```
struct dvb_frontend_ops {
    struct dvb_frontend_info info;
    u8 delsys[MAX_DELSYS];
    void (* release) (struct dvb_frontend* fe);
    void (* release_sec) (struct dvb_frontend* fe);
    int (* init) (struct dvb_frontend* fe);
    int (* sleep) (struct dvb_frontend* fe);
    int (* write) (struct dvb_frontend* fe, const u8 buf[], int len);
    int (* tune) (struct dvb_frontend* fe, bool re_tune, unsigned int mode_flags, unsigned int *freq);
    enum dvbfe_algo (* get_frontend_algo) (struct dvb_frontend *fe);
    int (* set_frontend) (struct dvb_frontend *fe);
    int (* get_tune_settings) (struct dvb_frontend* fe, struct dvb_frontend_tune_settings *tune_settings);
    int (* get_frontend) (struct dvb_frontend *fe);
    int (* read_status) (struct dvb_frontend *fe, enum fe_status *status);
    int (* read_ber) (struct dvb_frontend* fe, u32* ber);
    int (* read_signal_strength) (struct dvb_frontend* fe, u16* strength);
    int (* read_snr) (struct dvb_frontend* fe, u16* snr);
    int (* read_ucblocks) (struct dvb_frontend* fe, u32* ucblocks);
    int (* diseqc_reset_overload) (struct dvb_frontend* fe);
    int (* diseqc_send_master_cmd) (struct dvb_frontend* fe, struct dvb_diseqc_master_cmd *cmd);
    int (* diseqc_recv_slave_reply) (struct dvb_frontend* fe, struct dvb_diseqc_slave_reply *reply);
    int (* diseqc_send_burst) (struct dvb_frontend *fe, enum fe_sec_mini_cmd minicmd);
    int (* set_tone) (struct dvb_frontend *fe, enum fe_sec_tone_mode tone);
    int (* set_voltage) (struct dvb_frontend *fe, enum fe_sec_voltage voltage);
    int (* enable_high_lnb_voltage) (struct dvb_frontend* fe, long arg);
    int (* dishnetwork_send_legacy_command) (struct dvb_frontend* fe, unsigned long cmd);
    int (* i2c_gate_ctrl) (struct dvb_frontend* fe, int enable);
    int (* ts_bus_ctrl) (struct dvb_frontend* fe, int acquire);
    int (* set_lna) (struct dvb_frontend *);
    enum dvbfe_search (* search) (struct dvb_frontend *fe);
    struct dvb_tuner_ops tuner_ops;
    struct analog_demod_ops analog_ops;
    int (* set_property) (struct dvb_frontend* fe, struct dtv_property* tvp);
    int (* get_property) (struct dvb_frontend* fe, struct dtv_property* tvp);
};
```

## Members

info	embedded struct dvb_tuner_info with tuner properties
delsys[MAX_DELSYS]	Delivery systems supported by the frontend
release	callback function called when frontend is detached. drivers should free any allocated memory.
release_sec	callback function requesting that the Satellite Equipment Control (SEC) driver to release and free any memory allocated by the driver.

<code>init</code>	callback function used to initialize the tuner device.
<code>sleep</code>	callback function used to put the tuner to sleep.
<code>write</code>	callback function used by some demod legacy drivers to allow other drivers to write data into their registers. Should not be used on new drivers.
<code>tune</code>	callback function used by demod drivers that use <i>DVBFE_ALGO_HW</i> ; to tune into a frequency.
<code>get_frontend_algo</code>	returns the desired hardware algorithm.
<code>set_frontend</code>	callback function used to inform the demod to set the parameters for demodulating a digital TV channel. The properties to be used are stored at <i>dvb_frontend.dtv_property_cache</i> ; . The demod can change the parameters to reflect the changes needed for the channel to be decoded, and update statistics.
<code>get_tune_settings</code>	callback function
<code>get_frontend</code>	callback function used to inform the parameters actually in use. The properties to be used are stored at <i>dvb_frontend.dtv_property_cache</i> ; and update statistics. Please notice that it should not return an error code if the statistics are not available because the demog is not locked.
<code>read_status</code>	returns the locking status of the frontend.
<code>read_ber</code>	legacy callback function to return the bit error rate. Newer drivers should provide such info via DVBv5 API, e. g. <i>set_frontend</i> / <i>get_frontend</i> ; , implementing this callback only if DVBv3 API compatibility is wanted.
<code>read_signal_strength</code>	legacy callback function to return the signal strength. Newer drivers should provide such info via DVBv5 API, e. g. <i>set_frontend</i> / <i>get_frontend</i> ; , implementing this callback only if DVBv3 API compatibility is wanted.
<code>read_snr</code>	legacy callback function to return the Signal/Noise rate. Newer drivers should provide such info via DVBv5 API, e. g. <i>set_frontend</i> / <i>get_frontend</i> ; , implementing this callback only if DVBv3 API compatibility is wanted.
<code>read_ucblocks</code>	legacy callback function to return the Uncorrected Error Blocks. Newer drivers should provide such info via DVBv5 API, e. g. <i>set_frontend</i> / <i>get_frontend</i> ; , implementing this callback only if DVBv3 API compatibility is wanted.
<code>diseqc_reset_overload</code>	callback function to implement the <i>FE_DISEQC_RESET_OVERLOAD</i> ioctl (only Satellite)
<code>diseqc_send_master_cmd</code>	callback function to implement the <i>FE_DISEQC_SEND_MASTER_CMD</i> ioctl (only Satellite).
<code>diseqc_recv_slave_reply</code>	callback function to implement the <i>FE_DISEQC_RECV_SLAVE_REPLY</i> ioctl (only Satellite)

diseqc_send_burst	callback function to implement the FE_DISEQC_SEND_BURST ioctl (only Satellite).
set_tone	callback function to implement the FE_SET_TONE ioctl (only Satellite).
set_voltage	callback function to implement the FE_SET_VOLTAGE ioctl (only Satellite).
enable_high_lnb_voltage	callback function to implement the FE_ENABLE_HIGH_LNB_VOLTAGE ioctl (only Satellite).
dishnetwork_send_legacy_command	callback function to implement the FE_DISHNETWORK_SEND_LEGACY_CMD ioctl (only Satellite).
i2c_gate_ctrl	controls the I2C gate. Newer drivers should use I2C mux support instead.
ts_bus_ctrl	callback function used to take control of the TS bus.
set_lna	callback function to power on/off/auto the LNA.
search	callback function used on some custom algo search algos.
tuner_ops	pointer to struct dvb_tuner_ops
analog_ops	pointer to struct analog_demod_ops
set_property	callback function to allow the frontend to validate incoming properties. Should not be used on new drivers.
get_property	callback function to allow the frontend to override outgoing properties. Should not be used on new drivers.

## Name

struct dtv\_frontend\_properties — contains a list of properties that are specific to a digital TV standard.

## Synopsis

```
struct dtv_frontend_properties {
    u32 frequency;
    enum fe_modulation modulation;
    enum fe_sec_voltage voltage;
    enum fe_sec_tone_mode sectone;
    enum fe_spectral_inversion inversion;
    enum fe_code_rate fec_inner;
    enum fe_transmit_mode transmission_mode;
    u32 bandwidth_hz;
    enum fe_guard_interval guard_interval;
    enum fe_hierarchy hierarchy;
    u32 symbol_rate;
    enum fe_code_rate code_rate_HP;
    enum fe_code_rate code_rate_LP;
    enum fe_pilot pilot;
    enum fe_rolloff rolloff;
    enum fe_delivery_system delivery_system;
    enum fe_interleaving interleaving;
    u8 isdbt_partial_reception;
    u8 isdbt_sb_mode;
    u8 isdbt_sb_subchannel;
    u32 isdbt_sb_segment_idx;
    u32 isdbt_sb_segment_count;
    u8 isdbt_layer_enabled;
    struct layer[3];
    u32 stream_id;
    u8 atscmh_fic_ver;
    u8 atscmh_parade_id;
    u8 atscmh_nog;
    u8 atscmh_tnog;
    u8 atscmh_sgn;
    u8 atscmh_prc;
    u8 atscmh_rs_frame_mode;
    u8 atscmh_rs_frame_ensemble;
    u8 atscmh_rs_code_mode_pri;
    u8 atscmh_rs_code_mode_sec;
    u8 atscmh_sccc_block_mode;
    u8 atscmh_sccc_code_mode_a;
    u8 atscmh_sccc_code_mode_b;
    u8 atscmh_sccc_code_mode_c;
    u8 atscmh_sccc_code_mode_d;
    u32 lna;
    struct dtv_fe_stats strength;
    struct dtv_fe_stats cnr;
    struct dtv_fe_stats pre_bit_error;
    struct dtv_fe_stats pre_bit_count;
    struct dtv_fe_stats post_bit_error;
```

```
    struct dtv_fe_stats post_bit_count;
    struct dtv_fe_stats block_error;
    struct dtv_fe_stats block_count;
};
```

## Members

frequency	frequency in Hz for terrestrial/cable or in kHz for Satellite
modulation	Frontend modulation type
voltage	SEC voltage (only Satellite)
sectone	SEC tone mode (only Satellite)
inversion	Spectral inversion
fec_inner	Forward error correction inner Code Rate
transmission_mode	Transmission Mode
bandwidth_hz	Bandwidth, in Hz. A zero value means that userspace wants to autodetect.
guard_interval	Guard Interval
hierarchy	Hierarchy
symbol_rate	Symbol Rate
code_rate_HP	high priority stream code rate
code_rate_LP	low priority stream code rate
pilot	Enable/disable/autodetect pilot tones
rolloff	Rolloff factor (alpha)
delivery_system	FE delivery system (e. g. digital TV standard)
interleaving	interleaving
isdbt_partial_reception	ISDB-T partial reception (only ISDB standard)
isdbt_sb_mode	ISDB-T Sound Broadcast (SB) mode (only ISDB standard)
isdbt_sb_subchannel	ISDB-T SB subchannel (only ISDB standard)
isdbt_sb_segment_idx	ISDB-T SB segment index (only ISDB standard)
isdbt_sb_segment_count	ISDB-T SB segment count (only ISDB standard)
isdbt_layer_enabled	ISDB Layer enabled (only ISDB standard)
layer[3]	ISDB per-layer data (only ISDB standard) <i>layer.segment_count</i> : Segment Count; <i>layer.fec</i> : per layer code rate; <i>layer.modulation</i> : per layer modulation; <i>layer.interleaving</i> : per layer interleaving.



stream_id	If different than zero, enable substream filtering, if hardware supports (DVB-S2 and DVB-T2).
atscmh_fic_ver	Version number of the FIC (Fast Information Channel) signaling data (only ATSC-M/H)
atscmh_parade_id	Parade identification number (only ATSC-M/H)
atscmh_nog	Number of MH groups per MH subframe for a designated parade (only ATSC-M/H)
atscmh_tnog	Total number of MH groups including all MH groups belonging to all MH parades in one MH subframe (only ATSC-M/H)
atscmh_sgn	Start group number (only ATSC-M/H)
atscmh_prc	Parade repetition cycle (only ATSC-M/H)
atscmh_rs_frame_mode	Reed Solomon (RS) frame mode (only ATSC-M/H)
atscmh_rs_frame_ensemble	RS frame ensemble (only ATSC-M/H)
atscmh_rs_code_mode_pri	RS code mode pri (only ATSC-M/H)
atscmh_rs_code_mode_sec	RS code mode sec (only ATSC-M/H)
atscmh_sccc_block_mode	Series Concatenated Convolutional Code (SCCC) Block Mode (only ATSC-M/H)
atscmh_sccc_code_mode_a	SCCC code mode A (only ATSC-M/H)
atscmh_sccc_code_mode_b	SCCC code mode B (only ATSC-M/H)
atscmh_sccc_code_mode_c	SCCC code mode C (only ATSC-M/H)
atscmh_sccc_code_mode_d	SCCC code mode D (only ATSC-M/H)
lna	Power ON/OFF/AUTO the Linear Now-noise Amplifier (LNA)
strength	DVBv5 API statistics: Signal Strength
cnr	DVBv5 API statistics: Signal to Noise ratio of the (main) carrier
pre_bit_error	DVBv5 API statistics: pre-Viterbi bit error count
pre_bit_count	DVBv5 API statistics: pre-Viterbi bit count
post_bit_error	DVBv5 API statistics: post-Viterbi bit error count
post_bit_count	DVBv5 API statistics: post-Viterbi bit count
block_error	DVBv5 API statistics: block error count
block_count	DVBv5 API statistics: block count

## NOTE

derivated statistics like Uncorrected Error blocks (UCE) are calculated on userspace.

Only a subset of the properties are needed for a given delivery system. For more info, consult the [media\\_api.html](#) with the documentation of the Userspace API.

## Name

struct dvb\_frontend — Frontend structure to be used on drivers.

## Synopsis

```
struct dvb_frontend {
    struct dvb_frontend_ops ops;
    struct dvb_adapter * dvb;
    void * demodulator_priv;
    void * tuner_priv;
    void * frontend_priv;
    void * sec_priv;
    void * analog_demod_priv;
    struct dtv_frontend_properties dtv_property_cache;
#define DVB_FRONTEND_COMPONENT_TUNER 0
#define DVB_FRONTEND_COMPONENT_DEMOD 1
    int (* callback) (void *adapter_priv, int component, int cmd, int arg);
    int id;
    unsigned int exit;
};
```

## Members

ops	embedded struct dvb_frontend_ops
dvb	pointer to struct dvb_adapter
demodulator_priv	demod private data
tuner_priv	tuner private data
frontend_priv	frontend private data
sec_priv	SEC private data
analog_demod_priv	Analog demod private data
dtv_property_cache	embedded struct dtv_frontend_properties
callback	callback function used on some drivers to call either the tuner or the demodulator.
id	Frontend ID
exit	Used to inform the DVB core that the frontend thread should exit (usually, means that the hardware got disconnected).

## Name

`intlog2` — computes  $\log_2$  of a value; the result is shifted left by 24 bits

## Synopsis

```
unsigned int intlog2 (u32 value);
```

## Arguments

*value*    The value (must be  $\neq 0$ )

## to use rational values you can use the following method

$$\text{intlog2}(\text{value}) = \text{intlog2}(\text{value} * 2^x) - x * 2^{24}$$

## Some usecase examples

```
intlog2(8) will give  $3 \ll 24 = 3 * 2^{24}$   
intlog2(9) will give  $3 \ll 24 + \dots = 3.16\dots * 2^{24}$   
intlog2(1.5) =  $\text{intlog2}(3) - 2^{24} = 0.584\dots * 2^{24}$ 
```

## return

$$\log_2(\text{value}) * 2^{24}$$

## Name

`intlog10` — computes  $\log_{10}$  of a value; the result is shifted left by 24 bits

## Synopsis

```
unsigned int intlog10 (u32 value);
```

## Arguments

*value*    The value (must be  $\neq 0$ )

## to use rational values you can use the following method

$$\text{intlog10}(\text{value}) = \text{intlog10}(\text{value} * 10^x) - x * 2^{24}$$

## An usecase example

```
intlog10(1000) will give  $3 \ll 24 = 3 * 2^{24}$   
due to the implementation intlog10(1000) might be not exactly  $3 * 2^{24}$ 
```

look at `intlog2` for similar examples

## return

$$\log_{10}(\text{value}) * 2^{24}$$

## Name

`dvb_ringbuffer_pkt_write` — Write a packet into the ringbuffer.

## Synopsis

```
ssize_t dvb_ringbuffer_pkt_write (struct dvb_ringbuffer * rbuf, u8 *  
buf, size_t len);
```

## Arguments

*rbuf*    Ringbuffer to write to.

*buf*     Buffer to write.

*len*     Length of buffer (currently limited to 65535 bytes max). returns Number of bytes written, or -EFAULT, -ENOMEM, -EINVAL.

## Name

`dvb_ringbuffer_pkt_read_user` — Read from a packet in the ringbuffer.

## Synopsis

```
ssize_t dvb_ringbuffer_pkt_read_user (struct dvb_ringbuffer * rbuf,
size_t idx, int offset, u8 __user * buf, size_t len);
```

## Arguments

<i>rbuf</i>	Ringbuffer concerned.
<i>idx</i>	Packet index as returned by <code>dvb_ringbuffer_pkt_next</code> .
<i>offset</i>	Offset into packet to read from.
<i>buf</i>	Destination buffer for data.
<i>len</i>	Size of destination buffer.

## Note

unlike `dvb_ringbuffer_read`, this does NOT update the read pointer in the ringbuffer. You must use `dvb_ringbuffer_pkt_dispose` to mark a packet as no longer required.

## Description

returns Number of bytes read, or -EFAULT.

## Name

`dvb_ringbuffer_pkt_read` — Read from a packet in the ringbuffer.

## Synopsis

```
ssize_t dvb_ringbuffer_pkt_read (struct dvb_ringbuffer * rbuf, size_t
idx, int offset, u8 * buf, size_t len);
```

## Arguments

<i>rbuf</i>	Ringbuffer concerned.
<i>idx</i>	Packet index as returned by <code>dvb_ringbuffer_pkt_next</code> .
<i>offset</i>	Offset into packet to read from.
<i>buf</i>	Destination buffer for data.
<i>len</i>	Size of destination buffer.

## Note

unlike `dvb_ringbuffer_read_user`, this DOES update the read pointer in the ringbuffer.

## Description

returns Number of bytes read, or -EFAULT.



## Name

`dvb_ringbuffer_pkt_dispose` — Dispose of a packet in the ring buffer.

## Synopsis

```
void dvb_ringbuffer_pkt_dispose (struct dvb_ringbuffer * rbuf, size_t  
idx);
```

## Arguments

*rbuf*    Ring buffer concerned.

*idx*    Packet index as returned by `dvb_ringbuffer_pkt_next`.

## Name

`dvb_ringbuffer_pkt_next` — Get the index of the next packet in a ringbuffer.

## Synopsis

```
ssize_t dvb_ringbuffer_pkt_next (struct dvb_ringbuffer * rbuf, size_t  
idx, size_t * pktlen);
```

## Arguments

*rbuf*      Ringbuffer concerned.

*idx*        Previous packet index, or -1 to return the first packet index.

*pktlen*    On success, will be updated to contain the length of the packet in bytes. returns Packet index (if >=0), or -1 if no packets available.

## Name

struct dvb\_adapter — represents a Digital TV adapter using Linux DVB API

## Synopsis

```
struct dvb_adapter {
    int num;
    struct list_head list_head;
    struct list_head device_list;
    const char * name;
    u8 proposed_mac[6];
    void * priv;
    struct device * device;
    struct module * module;
    int mfe_shared;
    struct dvb_device * mfe_dvbdev;
    struct mutex mfe_lock;
#ifdef CONFIG_MEDIA_CONTROLLER_DVB
    struct media_device * mdev;
#endif
};
```

## Members

num	Number of the adapter
list_head	List with the DVB adapters
device_list	List with the DVB devices
name	Name of the adapter
proposed_mac[6]	proposed MAC address for the adapter
priv	private data
device	pointer to struct device
module	pointer to struct module
mfe_shared	mfe shared: indicates mutually exclusive frontends This usage of this flag is currently deprecated
mfe_dvbdev	Frontend device in use, in the case of MFE
mfe_lock	Lock to prevent using the other frontends when MFE is used.
mdev	pointer to struct media_device, used when the media controller is used.

## Name

struct dvb\_device — represents a DVB device node

## Synopsis

```
struct dvb_device {
    struct list_head list_head;
    const struct file_operations * fops;
    struct dvb_adapter * adapter;
    int type;
    int minor;
    u32 id;
    int readers;
    int writers;
    int users;
    wait_queue_head_t wait_queue;
    int (* kernel_ioctl) (struct file *file, unsigned int cmd, void *arg);
#ifdef CONFIG_MEDIA_CONTROLLER_DVB
    const char * name;
    struct media_entity * entity;
    struct media_pad * pads;
#endif
    void * priv;
};
```

## Members

list_head	List head with all DVB devices
fops	pointer to struct file_operations
adapter	pointer to the adapter that holds this device node
type	type of the device: DVB_DEVICE_SEC, DVB_DEVICE_FRONTEND, DVB_DEVICE_DEMUX, DVB_DEVICE_DVR, DVB_DEVICE_CA, DVB_DEVICE_NET
minor	devnode minor number. Major number is always DVB_MAJOR.
id	device ID number, inside the adapter
readers	Initialized by the caller. Each call to open in Read Only mode decreases this counter by one.
writers	Initialized by the caller. Each call to open in Read/Write mode decreases this counter by one.
users	Initialized by the caller. Each call to open in any mode decreases this counter by one.
wait_queue	wait queue, used to wait for certain events inside one of the DVB API callers
kernel_ioctl	callback function used to handle ioctl calls from userspace.
name	Name to be used for the device at the Media Controller

entity	pointer to struct media_entity associated with the device node
pads	pointer to struct media_pad associated with <i>entity</i> ;
priv	private data

## Description

This structure is used by the DVB core (frontend, CA, net, demux) in order to create the device nodes. Usually, driver should not initialize this struct directly.

## Name

`dvb_register_adapter` — Registers a new DVB adapter

## Synopsis

```
int dvb_register_adapter (struct dvb_adapter * adap, const char * name,  
struct module * module, struct device * device, short * adapter_nums);
```

## Arguments

<i>adap</i>	pointer to struct <code>dvb_adapter</code>
<i>name</i>	Adapter's name
<i>module</i>	initialized with <code>THIS_MODULE</code> at the caller
<i>device</i>	pointer to struct <code>device</code> that corresponds to the device driver
<i>adapter_nums</i>	Array with a list of the numbers for <code>dvb_register_adapter</code> ; to select among them. Typically, initialized with: <code>DVB_DEFINE_MOD_OPT_ADAPTER_NR(adapter_nums)</code>

## Name

`dvb_unregister_adapter` — Unregisters a DVB adapter

## Synopsis

```
int dvb_unregister_adapter (struct dvb_adapter * adap);
```

## Arguments

*adap* pointer to struct `dvb_adapter`

## Name

`dvb_register_device` — Registers a new DVB device

## Synopsis

```
int dvb_register_device (struct dvb_adapter * adap, struct dvb_device  
** pdvbdev, const struct dvb_device * template, void * priv, int type);
```

## Arguments

*adap*            pointer to struct `dvb_adapter`

*pdvbdev*       pointer to the place where the new struct `dvb_device` will be stored

*template*      Template used to create `pdvbdev`;

*priv*           private data

*type*           type of the device:    `DVB_DEVICE_SEC`,    `DVB_DEVICE_FRONTEND`,  
                 `DVB_DEVICE_DEMUX`,        `DVB_DEVICE_DVR`,        `DVB_DEVICE_CA`,  
                 `DVB_DEVICE_NET`



## Name

`dvb_unregister_device` — Unregisters a DVB device

## Synopsis

```
void dvb_unregister_device (struct dvb_device * dvbdev);
```

## Arguments

*dvbdev* pointer to struct `dvb_device`

# Digital TV Demux API

The kernel demux API defines a driver-internal interface for registering low-level, hardware specific driver to a hardware independent demux layer. It is only of interest for Digital TV device driver writers. The header file for this API is named `demux.h` and located in `drivers/media/dvb-core`.

The demux API should be implemented for each demux in the system. It is used to select the TS source of a demux and to manage the demux resources. When the demux client allocates a resource via the demux API, it receives a pointer to the API of that resource.

Each demux receives its TS input from a DVB front-end or from memory, as set via this demux API. In a system with more than one front-end, the API can be used to select one of the DVB front-ends as a TS source for a demux, unless this is fixed in the HW platform. The demux API only controls front-ends regarding to their connections with demuxes; the APIs used to set the other front-end parameters, such as tuning, are not defined in this document.

The functions that implement the abstract interface demux should be defined static or module private and registered to the Demux core for external access. It is not necessary to implement every function in the struct `dmx_demux`. For example, a demux interface might support Section filtering, but not PES filtering. The API client is expected to check the value of any function pointer before calling the function: the value of NULL means that the “function is not available”.

Whenever the functions of the demux API modify shared data, the possibilities of lost update and race condition problems should be addressed, e.g. by protecting parts of code with mutexes.

Note that functions called from a bottom half context must not sleep. Even a simple memory allocation without using `GFP_ATOMIC` can result in a kernel thread being put to sleep if swapping is needed. For example, the Linux kernel calls the functions of a network device interface from a bottom half context. Thus, if a demux API function is called from network device code, the function must not sleep.

## Demux Callback API

This kernel-space API comprises the callback functions that deliver filtered data to the demux client. Unlike the other DVB kABIs, these functions are provided by the client and called from the demux code.

The function pointers of this abstract interface are not packed into a structure as in the other demux APIs, because the callback functions are registered and used independent of each other. As an example, it is possible for the API client to provide several callback functions for receiving TS packets and no callbacks for PES packets or sections.

The functions that implement the callback API need not be re-entrant: when a demux driver calls one of these functions, the driver is not allowed to call the function again before the original call returns. If a

callback is triggered by a hardware interrupt, it is recommended to use the Linux “bottom half” mechanism or start a tasklet instead of making the callback function call directly from a hardware interrupt.

This mechanism is implemented by `dmx_ts_cb()` and `dmx_section_cb()`.

## Name

enum ts\_filter\_type — filter type bitmap for dmx\_ts\_feed.set

## Synopsis

```
enum ts_filter_type {  
    TS_PACKET,  
    TS_PAYLOAD_ONLY,  
    TS_DECODER,  
    TS_DEMUX  
};
```

## Constants

TS_PACKET	Send TS packets (188 bytes) to callback (default).
TS_PAYLOAD_ONLY	In case TS_PACKET is set, only send the TS payload (<=184 bytes per packet) to callback
TS_DECODER	Send stream to built-in decoder (if present).
TS_DEMUX	In case TS_PACKET is set, send the TS to the demux device, not to the dvr device

## Name

struct dmx\_ts\_feed — Structure that contains a TS feed filter

## Synopsis

```
struct dmx_ts_feed {
    int is_filtering;
    struct dmx_demux * parent;
    void * priv;
    int (* set) (struct dmx_ts_feed *feed, u16 pid, int type, enum dmx_ts_pes pes_type,
    int (* start_filtering) (struct dmx_ts_feed *feed);
    int (* stop_filtering) (struct dmx_ts_feed *feed);
};
```

## Members

is_filtering	Set to non-zero when filtering in progress
parent	pointer to struct dmx_demux
priv	pointer to private data of the API client
set	sets the TS filter
start_filtering	starts TS filtering
stop_filtering	stops TS filtering

## Description

A TS feed is typically mapped to a hardware PID filter on the demux chip. Using this API, the client can set the filtering properties to start/stop filtering TS packets on a particular TS feed.

## Name

struct dmxf\_section\_filter — Structure that describes a section filter

## Synopsis

```
struct dmxf_section_filter {  
    u8 filter_value[DMX_MAX_FILTER_SIZE];  
    u8 filter_mask[DMX_MAX_FILTER_SIZE];  
    u8 filter_mode[DMX_MAX_FILTER_SIZE];  
    struct dmxf_section_feed * parent;  
    void * priv;  
};
```

## Members

`filter_value[DMX_MAX_FILTER_SIZE]` contains up to 16 bytes (128 bits) of the TS section header that will be matched by the section filter

`filter_mask[DMX_MAX_FILTER_SIZE]` contains a 16 bytes (128 bits) filter mask with the bits specified by *filter\_value* that will be used on the filter match logic.

`filter_mode[DMX_MAX_FILTER_SIZE]` contains a 16 bytes (128 bits) filter mode.

`parent` Pointer to struct dmxf\_section\_feed.

`priv` Pointer to private data of the API client.

## Description

The *filter\_mask* controls which bits of *filter\_value* are compared with the section headers/payload. On a binary value of 1 in *filter\_mask*, the corresponding bits are compared. The filter only accepts sections that are equal to *filter\_value* in all the tested bit positions.

## Name

struct dmx\_section\_feed — Structure that contains a section feed filter

## Synopsis

```
struct dmx_section_feed {
    int is_filtering;
    struct dmx_demux * parent;
    void * priv;
    int check_crc;
    int (* set) (struct dmx_section_feed *feed, ul6 pid, size_t circular_buffer_size, i
    int (* allocate_filter) (struct dmx_section_feed *feed, struct dmx_section_filter
    int (* release_filter) (struct dmx_section_feed *feed, struct dmx_section_filter
    int (* start_filtering) (struct dmx_section_feed *feed);
    int (* stop_filtering) (struct dmx_section_feed *feed);
};
```

## Members

is_filtering	Set to non-zero when filtering in progress
parent	pointer to struct dmx_demux
priv	pointer to private data of the API client
check_crc	If non-zero, check the CRC values of filtered sections.
set	sets the section filter
allocate_filter	This function is used to allocate a section filter on the demux. It should only be called when no filtering is in progress on this section feed. If a filter cannot be allocated, the function fails with -ENOSPC.
release_filter	This function releases all the resources of a previously allocated section filter. The function should not be called while filtering is in progress on this section feed. After calling this function, the caller should not try to dereference the filter pointer.
start_filtering	starts section filtering
stop_filtering	stops section filtering

## Description

A TS feed is typically mapped to a hardware PID filter on the demux chip. Using this API, the client can set the filtering properties to start/stop filtering TS packets on a particular TS feed.

## Name

`dmx_ts_cb` — DVB demux TS filter callback function prototype

## Synopsis

```
int dmx_ts_cb (const u8 * buffer1, size_t buffer1_length, const u8 *  
buffer2, size_t buffer2_length, struct dmx_ts_feed * source);
```

## Arguments

<i>buffer1</i>	Pointer to the start of the filtered TS packets.
<i>buffer1_length</i>	Length of the TS data in <i>buffer1</i> .
<i>buffer2</i>	Pointer to the tail of the filtered TS packets, or NULL.
<i>buffer2_length</i>	Length of the TS data in <i>buffer2</i> .
<i>source</i>	Indicates which TS feed is the source of the callback.

## Description

This function callback prototype, provided by the client of the demux API, is called from the demux code. The function is only called when filtering on a TS feed has been enabled using the `start_filtering` function at the `dmx_demux`. Any TS packets that match the filter settings are copied to a circular buffer. The filtered TS packets are delivered to the client using this callback function. The size of the circular buffer is controlled by the `circular_buffer_size` parameter of the `dmx_ts_feed.set` function. It is expected that the *buffer1* and *buffer2* callback parameters point to addresses within the circular buffer, but other implementations are also possible. Note that the called party should not try to free the memory the *buffer1* and *buffer2* parameters point to.

When this function is called, the *buffer1* parameter typically points to the start of the first undelivered TS packet within a circular buffer. The *buffer2* buffer parameter is normally NULL, except when the received TS packets have crossed the last address of the circular buffer and "wrapped" to the beginning of the buffer. In the latter case the *buffer1* parameter would contain an address within the circular buffer, while the *buffer2* parameter would contain the first address of the circular buffer. The number of bytes delivered with this function (i.e. *buffer1\_length* + *buffer2\_length*) is usually equal to the value of `callback_length` parameter given in the `set` function, with one exception: if a timeout occurs before receiving `callback_length` bytes of TS data, any undelivered packets are immediately delivered to the client by calling this function. The timeout duration is controlled by the `set` function in the TS Feed API.

If a TS packet is received with errors that could not be fixed by the TS-level forward error correction (FEC), the `Transport_error_indicator` flag of the TS packet header should be set. The TS packet should not be discarded, as the error can possibly be corrected by a higher layer protocol. If the called party is slow in processing the callback, it is possible that the circular buffer eventually fills up. If this happens, the demux driver should discard any TS packets received while the buffer is full and return `-EOVERFLOW`.

The type of data returned to the callback can be selected by the `dmx_ts_feed.set` function. The `type` parameter decides if the raw TS packet (`TS_PACKET`) or just the payload (`TS_PACKET|TS_PAYLOAD_ONLY`) should be returned. If additionally the `TS_DECODER` bit is set the stream will also be sent to the hardware MPEG decoder.

## Return

0, on success; `-EOVERFLOW`, on buffer overflow.

## Name

`dmx_section_cb` — DVB demux TS filter callback function prototype

## Synopsis

```
int dmx_section_cb (const u8 * buffer1, size_t buffer1_len, const u8 *  
buffer2, size_t buffer2_len, struct dmx_section_filter * source);
```

## Arguments

<i>buffer1</i>	Pointer to the start of the filtered section, e.g. within the circular buffer of the demux driver.
<i>buffer1_len</i>	Length of the filtered section data in <i>buffer1</i> , including headers and CRC.
<i>buffer2</i>	Pointer to the tail of the filtered section data, or NULL. Useful to handle the wrapping of a circular buffer.
<i>buffer2_len</i>	Length of the filtered section data in <i>buffer2</i> , including headers and CRC.
<i>source</i>	Indicates which section feed is the source of the callback.

## Description

This function callback prototype, provided by the client of the demux API, is called from the demux code. The function is only called when filtering of sections has been enabled using the function `dmx_ts_feed.start_filtering`. When the demux driver has received a complete section that matches at least one section filter, the client is notified via this callback function. Normally this function is called for each received section; however, it is also possible to deliver multiple sections with one callback, for example when the system load is high. If an error occurs while receiving a section, this function should be called with the corresponding error type set in the success field, whether or not there is data to deliver. The Section Feed implementation should maintain a circular buffer for received sections. However, this is not necessary if the Section Feed API is implemented as a client of the TS Feed API, because the TS Feed implementation then buffers the received data. The size of the circular buffer can be configured using the `dmx_ts_feed.set` function in the Section Feed API. If there is no room in the circular buffer when a new section is received, the section must be discarded. If this happens, the value of the success parameter should be `DMX_OVERRUN_ERROR` on the next callback.



## Name

enum dmx\_frontend\_source — Used to identify the type of frontend

## Synopsis

```
enum dmx_frontend_source {  
    DMX_MEMORY_FE,  
    DMX_FRONTEND_0  
};
```

## Constants

DMX\_MEMORY\_FE The source of the demux is memory. It means that the MPEG-TS to be filtered comes from userspace, via `write` syscall.

DMX\_FRONTEND\_0 The source of the demux is a frontend connected to the demux.

## Name

struct dmx\_frontend — Structure that lists the frontends associated with a demux

## Synopsis

```
struct dmx_frontend {  
    struct list_head connectivity_list;  
    enum dmx_frontend_source source;  
};
```

## Members

connectivity_list	List of front-ends that can be connected to a particular demux;
source	Type of the frontend.

## FIXME

this structure should likely be replaced soon by some media-controller based logic.

## Name

enum dmx\_demux\_caps — MPEG-2 TS Demux capabilities bitmap

## Synopsis

```
enum dmx_demux_caps {  
    DMX_TS_FILTERING,  
    DMX_SECTION_FILTERING,  
    DMX_MEMORY_BASED_FILTERING  
};
```

## Constants

DMX_TS_FILTERING	set if TS filtering is supported;
DMX_SECTION_FILTERING	set if section filtering is supported;
DMX_MEMORY_BASED_FILTERING	if write available.

## Description

Those flags are OR'ed in the `dmx_demux.capabilities` field

## Name

struct dm<sub>x</sub>\_demux — Structure that contains the demux capabilities and callbacks.

## Synopsis

```
struct dmx_demux {
    enum dmx_demux_caps capabilities;
    struct dmx_frontend * frontend;
    void * priv;
    int (* open) (struct dmx_demux *demux);
    int (* close) (struct dmx_demux *demux);
    int (* write) (struct dmx_demux *demux, const char __user *buf, size_t count);
    int (* allocate_ts_feed) (struct dmx_demux *demux, struct dmx_ts_feed **feed, dmx_
    int (* release_ts_feed) (struct dmx_demux *demux, struct dmx_ts_feed *feed);
    int (* allocate_section_feed) (struct dmx_demux *demux, struct dmx_section_feed *f
    int (* release_section_feed) (struct dmx_demux *demux, struct dmx_section_feed *f
    int (* add_frontend) (struct dmx_demux *demux, struct dmx_frontend *frontend);
    int (* remove_frontend) (struct dmx_demux *demux, struct dmx_frontend *frontend);
    struct list_head *(* get_frontends) (struct dmx_demux *demux);
    int (* connect_frontend) (struct dmx_demux *demux, struct dmx_frontend *frontend);
    int (* disconnect_frontend) (struct dmx_demux *demux);
    int (* get_pes_pids) (struct dmx_demux *demux, ul6 *pids);
};
```

## Members

capabilities	Bitfield of capability flags.
frontend	Front-end connected to the demux
priv	Pointer to private data of the API client
open	This function reserves the demux for use by the caller and, if necessary, initializes the demux. When the demux is no longer needed, the function <i>close</i> should be called. It should be possible for multiple clients to access the demux at the same time. Thus, the function implementation should increment the demux usage count when <i>open</i> is called and decrement it when <i>close</i> is called. The <i>demux</i> function parameter contains a pointer to the demux API and instance data. It returns 0 on success; -EUSERS, if maximum usage count was reached; -EINVAL, on bad parameter.
close	This function reserves the demux for use by the caller and, if necessary, initializes the demux. When the demux is no longer needed, the function <i>close</i> should be called. It should be possible for multiple clients to access the demux at the same time. Thus, the function implementation should increment the demux usage count when <i>open</i> is called and decrement it when <i>close</i> is called. The <i>demux</i> function parameter contains a pointer to the demux API and instance data. It returns 0 on success; -ENODEV, if demux was not in use (e. g. no users); -EINVAL, on bad parameter.
write	This function provides the demux driver with a memory buffer containing TS packets. Instead of receiving TS packets from the DVB

front-end, the demux driver software will read packets from memory. Any clients of this demux with active TS, PES or Section filters will receive filtered data via the Demux callback API (see 0). The function returns when all the data in the buffer has been consumed by the demux. Demux hardware typically cannot read TS from memory. If this is the case, memory-based filtering has to be implemented entirely in software. The *demux* function parameter contains a pointer to the demux API and instance data. The *buf* function parameter contains a pointer to the TS data in kernel-space memory. The *count* function parameter contains the length of the TS data. It returns 0 on success; -ERESTARTSYS, if mutex lock was interrupted; -EINTR, if a signal handling is pending; -ENODEV, if demux was removed; -EINVAL, on bad parameter.

<code>allocate_ts_feed</code>	Allocates a new TS feed, which is used to filter the TS packets carrying a certain PID. The TS feed normally corresponds to a hardware PID filter on the demux chip. The <i>demux</i> function parameter contains a pointer to the demux API and instance data. The <i>feed</i> function parameter contains a pointer to the TS feed API and instance data. The <i>callback</i> function parameter contains a pointer to the callback function for passing received TS packet. It returns 0 on success; -ERESTARTSYS, if mutex lock was interrupted; -EBUSY, if no more TS feeds is available; -EINVAL, on bad parameter.
<code>release_ts_feed</code>	Releases the resources allocated with <i>allocate_ts_feed</i> . Any filtering in progress on the TS feed should be stopped before calling this function. The <i>demux</i> function parameter contains a pointer to the demux API and instance data. The <i>feed</i> function parameter contains a pointer to the TS feed API and instance data. It returns 0 on success; -EINVAL on bad parameter.
<code>allocate_section_feed</code>	Allocates a new section feed, i.e. a demux resource for filtering and receiving sections. On platforms with hardware support for section filtering, a section feed is directly mapped to the demux HW. On other platforms, TS packets are first PID filtered in hardware and a hardware section filter then emulated in software. The caller obtains an API pointer of type <code>dmx_section_feed_t</code> as an out parameter. Using this API the caller can set filtering parameters and start receiving sections. The <i>demux</i> function parameter contains a pointer to the demux API and instance data. The <i>feed</i> function parameter contains a pointer to the TS feed API and instance data. The <i>callback</i> function parameter contains a pointer to the callback function for passing received TS packet. It returns 0 on success; -EBUSY, if no more TS feeds is available; -EINVAL, on bad parameter.
<code>release_section_feed</code>	Releases the resources allocated with <i>allocate_section_feed</i> , including allocated filters. Any filtering in progress on the section feed should be stopped before calling this function. The <i>demux</i> function parameter contains a pointer to the demux API and instance data. The <i>feed</i> function parameter contains a pointer to the TS feed API and instance data. It returns 0 on success; -EINVAL, on bad parameter.
<code>add_frontend</code>	Registers a connectivity between a demux and a front-end, i.e., indicates that the demux can be connected via a call to <i>connect_frontend</i> to use the given front-end as a TS source. The client of this function

	<p>has to allocate dynamic or static memory for the frontend structure and initialize its fields before calling this function. This function is normally called during the driver initialization. The caller must not free the memory of the frontend struct before successfully calling <i>remove_frontend</i>. The <i>demux</i> function parameter contains a pointer to the demux API and instance data. The <i>frontend</i> function parameter contains a pointer to the front-end instance data. It returns 0 on success; -EINVAL, on bad parameter.</p>
<code>remove_frontend</code>	<p>Indicates that the given front-end, registered by a call to <i>add_frontend</i>, can no longer be connected as a TS source by this demux. The function should be called when a front-end driver or a demux driver is removed from the system. If the front-end is in use, the function fails with the return value of -EBUSY. After successfully calling this function, the caller can free the memory of the frontend struct if it was dynamically allocated before the <i>add_frontend</i> operation. The <i>demux</i> function parameter contains a pointer to the demux API and instance data. The <i>frontend</i> function parameter contains a pointer to the front-end instance data. It returns 0 on success; -ENODEV, if the front-end was not found, -EINVAL, on bad parameter.</p>
<code>get_frontends</code>	<p>Provides the APIs of the front-ends that have been registered for this demux. Any of the front-ends obtained with this call can be used as a parameter for <i>connect_frontend</i>. The include file <i>demux.h</i> contains the macro <i>DMX_FE_ENTRY</i> for converting an element of the generic type struct <i>list_head *</i> to the type struct <i>dmx_frontend *</i>. The caller must not free the memory of any of the elements obtained via this function call. The <i>demux</i> function parameter contains a pointer to the demux API and instance data. It returns a struct <i>list_head</i> pointer to the list of front-end interfaces, or NULL in the case of an empty list.</p>
<code>connect_frontend</code>	<p>Connects the TS output of the front-end to the input of the demux. A demux can only be connected to a front-end registered to the demux with the function <i>add_frontend</i>. It may or may not be possible to connect multiple demuxes to the same front-end, depending on the capabilities of the HW platform. When not used, the front-end should be released by calling <i>disconnect_frontend</i>. The <i>demux</i> function parameter contains a pointer to the demux API and instance data. The <i>frontend</i> function parameter contains a pointer to the front-end instance data. It returns 0 on success; -EINVAL, on bad parameter.</p>
<code>disconnect_frontend</code>	<p>Disconnects the demux and a front-end previously connected by a <i>connect_frontend</i> call. The <i>demux</i> function parameter contains a pointer to the demux API and instance data. It returns 0 on success; -EINVAL on bad parameter.</p>
<code>get_pes_pids</code>	<p>Get the PIDs for <i>DMX_PES_AUDIO0</i>, <i>DMX_PES_VIDEO0</i>, <i>DMX_PES_TELETEXT0</i>, <i>DMX_PES_SUBTITLE0</i> and <i>DMX_PES_PCR0</i>. The <i>demux</i> function parameter contains a pointer to the demux API and instance data. The <i>pids</i> function parameter contains an array with five u16 elements where the PIDs will be stored. It returns 0 on success; -EINVAL on bad parameter.</p>

## Remote Controller devices

## Name

struct rc\_scancode\_filter — Filter scan codes.

## Synopsis

```
struct rc_scancode_filter {  
    u32 data;  
    u32 mask;  
};
```

## Members

data     Scancode data to match.

mask     Mask of bits of scancode to compare.

## Name

enum rc\_filter\_type — Filter type constants.

## Synopsis

```
enum rc_filter_type {  
    RC_FILTER_NORMAL,  
    RC_FILTER_WAKEUP,  
    RC_FILTER_MAX  
};
```

## Constants

RC\_FILTER\_NORMAL Filter for normal operation.

RC\_FILTER\_WAKEUP Filter for waking from suspend.

RC\_FILTER\_MAX      Number of filter types.



## Name

struct rc\_dev — represents a remote control device

## Synopsis

```
struct rc_dev {
    struct device dev;
    const struct attribute_group * sysfs_groups[5];
    const char * input_name;
    const char * input_phys;
    struct input_id input_id;
    char * driver_name;
    const char * map_name;
    struct rc_map rc_map;
    struct mutex lock;
    unsigned int minor;
    struct ir_raw_event_ctrl * raw;
    struct input_dev * input_dev;
    enum rc_driver_type driver_type;
    bool idle;
    u64 allowed_protocols;
    u64 enabled_protocols;
    u64 allowed_wakeup_protocols;
    u64 enabled_wakeup_protocols;
    struct rc_scancode_filter scancode_filter;
    struct rc_scancode_filter scancode_wakeup_filter;
    u32 scancode_mask;
    u32 users;
    void * priv;
    spinlock_t keylock;
    bool keypressed;
    unsigned long keyup_jiffies;
    struct timer_list timer_keyup;
    u32 last_keycode;
    enum rc_type last_protocol;
    u32 last_scancode;
    u8 last_toggle;
    u32 timeout;
    u32 min_timeout;
    u32 max_timeout;
    u32 rx_resolution;
    u32 tx_resolution;
    int (* change_protocol) (struct rc_dev *dev, u64 *rc_type);
    int (* change_wakeup_protocol) (struct rc_dev *dev, u64 *rc_type);
    int (* open) (struct rc_dev *dev);
    void (* close) (struct rc_dev *dev);
    int (* s_tx_mask) (struct rc_dev *dev, u32 mask);
    int (* s_tx_carrier) (struct rc_dev *dev, u32 carrier);
    int (* s_tx_duty_cycle) (struct rc_dev *dev, u32 duty_cycle);
    int (* s_rx_carrier_range) (struct rc_dev *dev, u32 min, u32 max);
    int (* tx_ir) (struct rc_dev *dev, unsigned *txbuf, unsigned n);
    void (* s_idle) (struct rc_dev *dev, bool enable);
```

```
int (* s_learning_mode) (struct rc_dev *dev, int enable);
int (* s_carrier_report) (struct rc_dev *dev, int enable);
int (* s_filter) (struct rc_dev *dev, struct rc_scancode_filter *filter);
int (* s_wakeup_filter) (struct rc_dev *dev, struct rc_scancode_filter *filter);
};
```

## Members

dev	driver model's view of this device
sysfs_groups[5]	sysfs attribute groups
input_name	name of the input child device
input_phys	physical path to the input child device
input_id	id of the input child device (struct input_id)
driver_name	name of the hardware driver which registered this device
map_name	name of the default keymap
rc_map	current scan/key table
lock	used to ensure we've filled in all protocol details before anyone can call show_protocols or store_protocols
minor	unique minor remote control device number
raw	additional data for raw pulse/space devices
input_dev	the input child device used to communicate events to userspace
driver_type	specifies if protocol decoding is done in hardware or software
idle	used to keep track of RX state
allowed_protocols	bitmask with the supported RC_BIT_* protocols
enabled_protocols	bitmask with the enabled RC_BIT_* protocols
allowed_wakeup_protocols	bitmask with the supported RC_BIT_* wakeup protocols
enabled_wakeup_protocols	bitmask with the enabled RC_BIT_* wakeup protocols
scancode_filter	scancode filter
scancode_wakeup_filter	scancode wakeup filters
scancode_mask	some hardware decoders are not capable of providing the full scancode to the application. As this is a hardware limit, we can't do anything with it. Yet, as the same keycode table can be used with other devices, a mask is provided to allow its usage. Drivers should generally leave this field in blank
users	number of current users of the device
priv	driver-specific data

keylock	protects the remaining members of the struct
keypressed	whether a key is currently pressed
keyup_jiffies	time (in jiffies) when the current keypress should be released
timer_keyup	timer for releasing a keypress
last_keycode	keycode of last keypress
last_protocol	protocol of last keypress
last_scancode	scancode of last keypress
last_toggle	toggle value of last command
timeout	optional time after which device stops sending data
min_timeout	minimum timeout supported by device
max_timeout	maximum timeout supported by device
rx_resolution	resolution (in ns) of input sampler
tx_resolution	resolution (in ns) of output sampler
change_protocol	allow changing the protocol used on hardware decoders
change_wakeup_protocol	allow changing the protocol used for wakeup filtering
open	callback to allow drivers to enable polling/irq when IR input device is opened.
close	callback to allow drivers to disable polling/irq when IR input device is opened.
s_tx_mask	set transmitter mask (for devices with multiple tx outputs)
s_tx_carrier	set transmit carrier frequency
s_tx_duty_cycle	set transmit duty cycle (0% - 100%)
s_rx_carrier_range	inform driver about carrier it is expected to handle
tx_ir	transmit IR
s_idle	enable/disable hardware idle mode, upon which, device doesn't interrupt host until it sees IR pulses
s_learning_mode	enable wide band receiver used for learning
s_carrier_report	enable carrier reports
s_filter	set the scancode filter
s_wakeup_filter	set the wakeup scancode filter

## Name

struct lirc\_driver — Defines the parameters on a LIRC driver

## Synopsis

```
struct lirc_driver {
    char name[40];
    int minor;
    __u32 code_length;
    unsigned int buffer_size;
    int sample_rate;
    __u32 features;
    unsigned int chunk_size;
    void * data;
    int min_timeout;
    int max_timeout;
    int (* add_to_buf) (void *data, struct lirc_buffer *buf);
    struct lirc_buffer * rbuf;
    int (* set_use_inc) (void *data);
    void (* set_use_dec) (void *data);
    struct rc_dev * rdev;
    const struct file_operations * fops;
    struct device * dev;
    struct module * owner;
};
```

## Members

name[40]	this string will be used for logs
minor	indicates minor device (/dev/lirc) number for registered driver if caller fills it with negative value, then the first free minor number will be used (if available).
code_length	length of the remote control key code expressed in bits.
buffer_size	Number of FIFO buffers with <i>chunk_size</i> size. If zero, creates a buffer with BUFLen size (16 bytes).
sample_rate	if zero, the device will wait for an event with a new code to be parsed. Otherwise, specifies the sample rate for polling. Value should be between 0 and HZ. If equal to HZ, it would mean one polling per second.
features	lirc compatible hardware features, like LIRC_MODE_RAW, LIRC_CAN_*, as defined at include/media/lirc.h.
chunk_size	Size of each FIFO buffer.
data	it may point to any driver data and this pointer will be passed to all callback functions.
min_timeout	Minimum timeout for record. Valid only if LIRC_CAN_SET_REC_TIMEOUT is defined.
max_timeout	Maximum timeout for record. Valid only if LIRC_CAN_SET_REC_TIMEOUT is defined.

<code>add_to_buf</code>	<code>add_to_buf</code> will be called after specified period of the time or triggered by the external event, this behavior depends on value of the <code>sample_rate</code> this function will be called in user context. This routine should return 0 if data was added to the buffer and <code>-ENODATA</code> if none was available. This should add some number of bits evenly divisible by <code>code_length</code> to the buffer.
<code>rbuf</code>	if not <code>NULL</code> , it will be used as a read buffer, you will have to write to the buffer by other means, like <code>irq's</code> (see also <code>lirc_serial.c</code> ).
<code>set_use_inc</code>	<code>set_use_inc</code> will be called after device is opened
<code>set_use_dec</code>	<code>set_use_dec</code> will be called after device is closed
<code>rdev</code>	Pointed to struct <code>rc_dev</code> associated with the LIRC device.
<code>fops</code>	<code>file_operations</code> for drivers which don't fit the current driver model. Some <code>ioctl's</code> can be directly handled by <code>lirc_dev</code> if the driver's <code>ioctl</code> function is <code>NULL</code> or if it returns <code>-ENOIOCTLCMD</code> (see also <code>lirc_serial.c</code> ).
<code>dev</code>	pointer to the struct device associated with the LIRC device.
<code>owner</code>	the module owning this struct

## Media Controller devices

## Name

struct media\_device — Media device

## Synopsis

```
struct media_device {
    struct device * dev;
    struct media_devnode devnode;
    char model[32];
    char serial[40];
    char bus_info[32];
    u32 hw_revision;
    u32 driver_version;
    u32 entity_id;
    struct list_head entities;
    spinlock_t lock;
    struct mutex graph_mutex;
    int (* link_notify) (struct media_link *link, u32 flags,unsigned int notificatio
};
```

## Members

dev	Parent device
devnode	Media device node
model[32]	Device model name
serial[40]	Device serial number (optional)
bus_info[32]	Unique and stable device location identifier
hw_revision	Hardware device revision
driver_version	Device driver version
entity_id	ID of the next entity to be registered
entities	List of registered entities
lock	Entities list lock
graph_mutex	Entities graph operation lock
link_notify	Link state change notification callback

## Description

This structure represents an abstract high-level media device. It allows easy access to entities and provides basic media device-level support. The structure can be allocated directly or embedded in a larger structure.

The parent *dev* is a physical device. It must be set before registering the media device.

*model* is a descriptive model name exported through sysfs. It doesn't have to be unique.

## Name

struct media\_devnode — Media device node

## Synopsis

```
struct media_devnode {
    const struct media_file_operations * fops;
    struct device dev;
    struct cdev cdev;
    struct device * parent;
    int minor;
    unsigned long flags;
    void (* release) (struct media_devnode *mdev);
};
```

## Members

fops	pointer to struct media_file_operations with media device ops
dev	struct device pointer for the media controller device
cdev	struct cdev pointer character device
parent	parent device
minor	device node minor number
flags	flags, combination of the MEDIA_FLAG_* constants
release	release callback called at the end of media_devnode_release

## Description

This structure represents a media-related device node.

The *parent* is a physical device. It must be set by core or device drivers before registering the node.

## Name

struct media\_entity\_operations — Media entity operations

## Synopsis

```
struct media_entity_operations {  
    int (* link_setup) (struct media_entity *entity, const struct media_pad *local, const struct media_pad *remote);  
    int (* link_validate) (struct media_link *link);  
};
```

## Members

link_setup	Notify the entity of link changes. The operation can return an error, in which case link setup will be cancelled. Optional.
link_validate	Return whether a link is valid from the entity point of view. The <code>media_entity_pipeline_start</code> function validates all links by calling this operation. Optional.



---

## Chapter 7. 16x50 UART Driver

## Name

`uart_update_timeout` — update per-port FIFO timeout.

## Synopsis

```
void uart_update_timeout (struct uart_port * port, unsigned int cflag,  
unsigned int baud);
```

## Arguments

*port*     `uart_port` structure describing the port

*cflag*    `termios cflag` value

*baud*     speed of the port

## Description

Set the port FIFO timeout value. The *cflag* value should reflect the actual hardware settings.

## Name

`uart_get_baud_rate` — return baud rate for a particular port

## Synopsis

```
unsigned int uart_get_baud_rate (struct uart_port * port, struct
ktermios * termios, struct ktermios * old, unsigned int min, unsigned
int max);
```

## Arguments

<i>port</i>	uart_port structure describing the port in question.
<i>termios</i>	desired termios settings.
<i>old</i>	old termios (or NULL)
<i>min</i>	minimum acceptable baud rate
<i>max</i>	maximum acceptable baud rate

## Description

Decode the termios structure into a numeric baud rate, taking account of the magic 38400 baud rate (with `spd_*` flags), and mapping the B0 rate to 9600 baud.

If the new baud rate is invalid, try the old termios setting. If it's still invalid, we try 9600 baud.

Update the *termios* structure to reflect the baud rate we're actually going to be using. Don't do this for the case where B0 is requested (“hang up”).

## Name

`uart_get_divisor` — return uart clock divisor

## Synopsis

```
unsigned int uart_get_divisor (struct uart_port * port, unsigned int
baud);
```

## Arguments

*port* uart\_port structure describing the port.

*baud* desired baud rate

## Description

Calculate the uart clock divisor for the port.

## Name

`uart_console_write` — write a console message to a serial port

## Synopsis

```
void uart_console_write (struct uart_port * port, const char * s,  
unsigned int count, void (*putchar) (struct uart_port *, int));
```

## Arguments

<i>port</i>	the port to write the message
<i>s</i>	array of characters
<i>count</i>	number of characters in string to write
<i>putchar</i>	function to write character to port

## Name

uart\_parse\_earlycon — Parse earlycon options

## Synopsis

```
int uart_parse_earlycon (char * p, unsigned char * iotype, unsigned long  
* addr, char ** options);
```

## Arguments

*p* ptr to 2nd field (ie., just beyond '<name>,')  
*iotype* ptr for decoded iotype (out)  
*addr* ptr for decoded mapbase/iobase (out)  
*options* ptr for <options> field; NULL if not present (out)

## Description

Decodes earlycon kernel command line parameters of the form earlycon=<name>,io|mmio|mmio32|mmio32be|mmio32native,<addr>,<options> console=<name>,io|mmio|mmio32|mmio32be|mmio32native,<addr>,<options>

The optional form earlycon=<name>,0x<addr>,<options> console=<name>,0x<addr>,<options> is also accepted; the returned *iotype* will be UPIO\_MEM.

Returns 0 on success or -EINVAL on failure

## Name

`uart_parse_options` — Parse serial port baud/parity/bits/flow control.

## Synopsis

```
void uart_parse_options (char * options, int * baud, int * parity, int  
* bits, int * flow);
```

## Arguments

*options*   pointer to option string

*baud*       pointer to an 'int' variable for the baud rate.

*parity*    pointer to an 'int' variable for the parity.

*bits*       pointer to an 'int' variable for the number of data bits.

*flow*       pointer to an 'int' variable for the flow control character.

## Description

`uart_parse_options` decodes a string containing the serial console options. The format of the string is `<baud><parity><bits><flow>`,

## eg

115200n8r

## Name

`uart_set_options` — setup the serial console parameters

## Synopsis

```
int uart_set_options (struct uart_port * port, struct console * co, int
baud, int parity, int bits, int flow);
```

## Arguments

<i>port</i>	pointer to the serial ports <code>uart_port</code> structure
<i>co</i>	console pointer
<i>baud</i>	baud rate
<i>parity</i>	parity character - 'n' (none), 'o' (odd), 'e' (even)
<i>bits</i>	number of data bits
<i>flow</i>	flow control character - 'r' (rts)



## Name

`uart_register_driver` — register a driver with the uart core layer

## Synopsis

```
int uart_register_driver (struct uart_driver * drv);
```

## Arguments

*drv*    low level driver structure

## Description

Register a uart driver with the core driver. We in turn register with the tty layer, and initialise the core driver per-port state.

We have a proc file in `/proc/tty/driver` which is named after the normal driver.

`drv->port` should be NULL, and the per-port structures should be registered using `uart_add_one_port` after this call has succeeded.

## Name

`uart_unregister_driver` — remove a driver from the uart core layer

## Synopsis

```
void uart_unregister_driver (struct uart_driver * drv);
```

## Arguments

*drv*    low level driver structure

## Description

Remove all references to a driver from the core driver. The low level driver must have removed all its ports via the `uart_remove_one_port` if it registered them with `uart_add_one_port`. (ie, `drv->port == NULL`)

## Name

`uart_add_one_port` — attach a driver-defined port structure

## Synopsis

```
int uart_add_one_port (struct uart_driver * drv, struct uart_port *  
uport);
```

## Arguments

*drv* pointer to the uart low level driver structure for this port

*uport* uart port structure to use for this port.

## Description

This allows the driver to register its own `uart_port` structure with the core driver. The main purpose is to allow the low level uart drivers to expand `uart_port`, rather than having yet more levels of structures.

## Name

`uart_remove_one_port` — detach a driver defined port structure

## Synopsis

```
int uart_remove_one_port (struct uart_driver * drv, struct uart_port  
* uport);
```

## Arguments

*drv* pointer to the uart low level driver structure for this port

*uport* uart port structure for this port

## Description

This unhooks (and hangs up) the specified port structure from the core driver. No further calls will be made to the low-level code for this port.

## Name

`uart_handle_dcd_change` — handle a change of carrier detect state

## Synopsis

```
void uart_handle_dcd_change (struct uart_port * uport, unsigned int  
status);
```

## Arguments

*uport*    `uart_port` structure for the open port

*status*   new carrier detect status, nonzero if active

## Description

Caller must hold `uport->lock`

## Name

`uart_handle_cts_change` — handle a change of clear-to-send state

## Synopsis

```
void uart_handle_cts_change (struct uart_port * uport, unsigned int  
status);
```

## Arguments

*uport*     `uart_port` structure for the open port

*status*    new clear to send status, nonzero if active

## Description

Caller must hold `uport->lock`

## Name

`uart_insert_char` — push a char to the uart layer

## Synopsis

```
void uart_insert_char (struct uart_port * port, unsigned int status,  
unsigned int overrun, unsigned int ch, unsigned int flag);
```

## Arguments

<i>port</i>	corresponding port
<i>status</i>	state of the serial port RX buffer (LSR for 8250)
<i>overrun</i>	mask of overrun bits in <i>status</i>
<i>ch</i>	character to push
<i>flag</i>	flag for the character (see TTY_NORMAL and friends)

## Description

User is responsible to call `tty_flip_buffer_push` when they are done with insertion.

## Name

serial8250\_get\_port — retrieve struct uart\_8250\_port

## Synopsis

```
struct uart_8250_port * serial8250_get_port (int line);
```

## Arguments

*line* serial line number

## Description

This function retrieves struct uart\_8250\_port for the specific line. This struct *must not* be used to perform a 8250 or serial core operation which is not accessible otherwise. Its only purpose is to make the struct accessible to the runtime-pm callbacks for context suspend/restore. The lock assumption made here is none because runtime-pm suspend/resume callbacks should not be invoked if there is any operation performed on the port.



## Name

`serial8250_suspend_port` — suspend one serial port

## Synopsis

```
void serial8250_suspend_port (int line);
```

## Arguments

*line* serial line number

## Description

Suspend one serial port.

## Name

`serial8250_resume_port` — resume one serial port

## Synopsis

```
void serial8250_resume_port (int line);
```

## Arguments

*line*    serial line number

## Description

Resume one serial port.

## Name

`serial8250_register_8250_port` — register a serial port

## Synopsis

```
int serial8250_register_8250_port (struct uart_8250_port * up);
```

## Arguments

*up* serial port template

## Description

Configure the serial port specified by the request. If the port exists and is in use, it is hung up and unregistered first.

The port is then probed and if necessary the IRQ is autodetected. If this fails an error is returned.

On success the port is ready to use and the line number is returned.

## Name

`serial8250_unregister_port` — remove a 16x50 serial port at runtime

## Synopsis

```
void serial8250_unregister_port (int line);
```

## Arguments

*line* serial line number

## Description

Remove one serial port. This may not be called from interrupt context. We hand the port back to the our control.

---

# Chapter 8. Frame Buffer Library

The frame buffer drivers depend heavily on four data structures. These structures are declared in `include/linux/fb.h`. They are `fb_info`, `fb_var_screeninfo`, `fb_fix_screeninfo` and `fb_monospecs`. The last three can be made available to and from userland.

`fb_info` defines the current state of a particular video card. Inside `fb_info`, there exists a `fb_ops` structure which is a collection of needed functions to make `fbdev` and `fbcon` work. `fb_info` is only visible to the kernel.

`fb_var_screeninfo` is used to describe the features of a video card that are user defined. With `fb_var_screeninfo`, things such as depth and the resolution may be defined.

The next structure is `fb_fix_screeninfo`. This defines the properties of a card that are created when a mode is set and can't be changed otherwise. A good example of this is the start of the frame buffer memory. This "locks" the address of the frame buffer memory, so that it cannot be changed or moved.

The last structure is `fb_monospecs`. In the old API, there was little importance for `fb_monospecs`. This allowed for forbidden things such as setting a mode of 800x600 on a fix frequency monitor. With the new API, `fb_monospecs` prevents such things, and if used correctly, can prevent a monitor from being cooked. `fb_monospecs` will not be useful until kernels 2.5.x.

## Frame Buffer Memory

## Name

`register_framebuffer` — registers a frame buffer device

## Synopsis

```
int register_framebuffer (struct fb_info * fb_info);
```

## Arguments

*fb\_info* frame buffer info structure

## Description

Registers a frame buffer device *fb\_info*.

Returns negative `errno` on error, or zero for success.

## Name

`unregister_framebuffer` — releases a frame buffer device

## Synopsis

```
int unregister_framebuffer (struct fb_info * fb_info);
```

## Arguments

*fb\_info* frame buffer info structure

## Description

Unregisters a frame buffer device *fb\_info*.

Returns negative `errno` on error, or zero for success.

This function will also notify the framebuffer console to release the driver.

This is meant to be called within a driver's `module_exit` function. If this is called outside `module_exit`, ensure that the driver implements `fb_open` and `fb_release` to check that no processes are using the device.

## Name

`fb_set_suspend` — low level driver signals suspend

## Synopsis

```
void fb_set_suspend (struct fb_info * info, int state);
```

## Arguments

*info*    framebuffer affected

*state*   0 = resuming, !=0 = suspending

## Description

This is meant to be used by low level drivers to signal suspend/resume to the core & clients. It must be called with the console semaphore held

# Frame Buffer Colormap



## Name

`fb_dealloc_cmap` — deallocate a colormap

## Synopsis

```
void fb_dealloc_cmap (struct fb_cmap * cmap);
```

## Arguments

*cmap*    frame buffer colormap structure

## Description

Deallocates a colormap that was previously allocated with `fb_alloc_cmap`.

## Name

`fb_copy_cmap` — copy a colormap

## Synopsis

```
int fb_copy_cmap (const struct fb_cmap * from, struct fb_cmap * to);
```

## Arguments

*from*    frame buffer colormap structure

*to*      frame buffer colormap structure

## Description

Copy contents of colormap from *from* to *to*.

## Name

`fb_set_cmap` — set the colormap

## Synopsis

```
int fb_set_cmap (struct fb_cmap * cmap, struct fb_info * info);
```

## Arguments

*cmap* frame buffer colormap structure

*info* frame buffer info structure

## Description

Sets the colormap *cmap* for a screen of device *info*.

Returns negative `errno` on error, or zero on success.

## Name

`fb_default_cmap` — get default colormap

## Synopsis

```
const struct fb_cmap * fb_default_cmap (int len);
```

## Arguments

*len*    size of palette for a depth

## Description

Gets the default colormap for a specific screen depth. *len* is the size of the palette for a particular screen depth.

Returns pointer to a frame buffer colormap structure.

## Name

`fb_invert_cmaps` — invert all defaults colormaps

## Synopsis

```
void fb_invert_cmaps ( void );
```

## Arguments

*void* no arguments

## Description

Invert all default colormaps.

# Frame Buffer Video Mode Database

## Name

`fb_try_mode` — test a video mode

## Synopsis

```
int fb_try_mode (struct fb_var_screeninfo * var, struct fb_info * info,  
const struct fb_videomode * mode, unsigned int bpp);
```

## Arguments

*var*    frame buffer user defined part of display

*info*   frame buffer info structure

*mode*   frame buffer video mode structure

*bpp*    color depth in bits per pixel

## Description

Tries a video mode to test it's validity for device *info*.

Returns 1 on success.

## Name

`fb_delete_videomode` — removed videomode entry from modelist

## Synopsis

```
void fb_delete_videomode (const struct fb_videomode * mode, struct  
list_head * head);
```

## Arguments

*mode* videomode to remove

*head* struct list\_head of modelist

## NOTES

Will remove all matching mode entries

## Name

`fb_find_mode` — finds a valid video mode

## Synopsis

```
int fb_find_mode (struct fb_var_screeninfo * var, struct fb_info * info,
const char * mode_option, const struct fb_videomode * db, unsigned
int dbsize, const struct fb_videomode * default_mode, unsigned int
default_bpp);
```

## Arguments

<i>var</i>	frame buffer user defined part of display
<i>info</i>	frame buffer info structure
<i>mode_option</i>	string video mode to find
<i>db</i>	video mode database
<i>dbsize</i>	size of <i>db</i>
<i>default_mode</i>	default video mode to fall back to
<i>default_bpp</i>	default color depth in bits per pixel

## Description

Finds a suitable video mode, starting with the specified mode in *mode\_option* with fallback to *default\_mode*. If *default\_mode* fails, all modes in the video mode database will be tried.

Valid mode specifiers for *mode\_option*:

<xres>x<yres>[M][R][-<bpp>][@<refresh>][i][m] or <name>[-<bpp>][@<refresh>]

with <xres>, <yres>, <bpp> and <refresh> decimal numbers and <name> a string.

If 'M' is present after yres (and before refresh/bpp if present), the function will compute the timings using VESA(tm) Coordinated Video Timings (CVT). If 'R' is present after 'M', will compute with reduced blanking (for flatpanels). If 'i' is present, compute interlaced mode. If 'm' is present, add margins equal to 1.8% of xres rounded down to 8 pixels, and 1.8% of yres. The char 'i' and 'm' must be after 'M' and 'R'. Example:

1024x768MR-860m - Reduced blank with margins at 60Hz.

## NOTE

The passed struct *var* is `_not_` cleared! This allows you to supply values for e.g. the grayscale and `accel_flags` fields.

Returns zero for failure, 1 if using specified *mode\_option*, 2 if using specified *mode\_option* with an ignored refresh rate, 3 if default mode is used, 4 if fall back to any valid mode.



## Name

`fb_var_to_videomode` — convert `fb_var_screeninfo` to `fb_videomode`

## Synopsis

```
void fb_var_to_videomode (struct fb_videomode * mode, const struct  
fb_var_screeninfo * var);
```

## Arguments

*mode* pointer to struct `fb_videomode`

*var* pointer to struct `fb_var_screeninfo`

## Name

`fb_videomode_to_var` — convert `fb_videomode` to `fb_var_screeninfo`

## Synopsis

```
void fb_videomode_to_var (struct fb_var_screeninfo * var, const struct  
fb_videomode * mode);
```

## Arguments

*var*    pointer to struct `fb_var_screeninfo`

*mode*   pointer to struct `fb_videomode`

## Name

`fb_mode_is_equal` — compare 2 videomodes

## Synopsis

```
int fb_mode_is_equal (const struct fb_videomode * mode1, const struct  
fb_videomode * mode2);
```

## Arguments

*mode1*   first videomode

*mode2*   second videomode

## RETURNS

1 if equal, 0 if not

## Name

`fb_find_best_mode` — find best matching videomode

## Synopsis

```
const struct fb_videomode * fb_find_best_mode (const struct
fb_var_screeninfo * var, struct list_head * head);
```

## Arguments

*var* pointer to struct `fb_var_screeninfo`

*head* pointer to struct `list_head` of modelist

## RETURNS

struct `fb_videomode`, NULL if none found

## IMPORTANT

This function assumes that all modelist entries in `info->modelist` are valid.

## NOTES

Finds best matching videomode which has an equal or greater dimension than `var->xres` and `var->yres`. If more than 1 videomode is found, will return the videomode with the highest refresh rate

## Name

`fb_find_nearest_mode` — find closest videomode

## Synopsis

```
const struct fb_videomode * fb_find_nearest_mode (const struct
fb_videomode * mode, struct list_head * head);
```

## Arguments

*mode* pointer to struct fb\_videomode

*head* pointer to modelist

## Description

Finds best matching videomode, smaller or greater in dimension. If more than 1 videomode is found, will return the videomode with the closest refresh rate.

## Name

`fb_match_mode` — find a videomode which exactly matches the timings in `var`

## Synopsis

```
const struct fb_videomode * fb_match_mode (const struct
fb_var_screeninfo * var, struct list_head * head);
```

## Arguments

*var* pointer to struct `fb_var_screeninfo`

*head* pointer to struct `list_head` of modelist

## RETURNS

struct `fb_videomode`, NULL if none found

## Name

`fb_add_videomode` — adds videomode entry to modelist

## Synopsis

```
int fb_add_videomode (const struct fb_videomode * mode, struct list_head  
* head);
```

## Arguments

*mode* videomode to add

*head* struct list\_head of modelist

## NOTES

Will only add unmatched mode entries

## Name

`fb_destroy_modelist` — destroy modelist

## Synopsis

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

## Arguments

*head*   struct list\_head of modelist



## Name

`fb_videomode_to_modelist` — convert mode array to mode list

## Synopsis

```
void fb_videomode_to_modelist (const struct fb_videomode * modedb, int
    num, struct list_head * head);
```

## Arguments

*modedb*    array of struct fb\_videomode

*num*        number of entries in array

*head*       struct list\_head of modelist

# Frame Buffer Macintosh Video Mode Database

## Name

`mac_vmode_to_var` — converts vmode/cmode pair to var structure

## Synopsis

```
int mac_vmode_to_var (int vmode, int cmode, struct fb_var_screeninfo  
* var);
```

## Arguments

*vmode*    MacOS video mode

*cmode*    MacOS color mode

*var*      frame buffer video mode structure

## Description

Converts a MacOS vmode/cmode pair to a frame buffer video mode structure.

Returns negative errno on error, or zero for success.

## Name

`mac_map_monitor_sense` — Convert monitor sense to vmode

## Synopsis

```
int mac_map_monitor_sense (int sense);
```

## Arguments

*sense*    Macintosh monitor sense number

## Description

Converts a Macintosh monitor sense number to a MacOS vmode number.

Returns MacOS vmode video mode number.

## Name

`mac_find_mode` — find a video mode

## Synopsis

```
int mac_find_mode (struct fb_var_screeninfo * var, struct fb_info *
info, const char * mode_option, unsigned int default_bpp);
```

## Arguments

<i>var</i>	frame buffer user defined part of display
<i>info</i>	frame buffer info structure
<i>mode_option</i>	video mode name (see <code>mac_modedb[]</code> )
<i>default_bpp</i>	default color depth in bits per pixel

## Description

Finds a suitable video mode. Tries to set mode specified by *mode\_option*. If the name of the wanted mode begins with 'mac', the Mac video mode database will be used, otherwise it will fall back to the standard video mode database.

## Note

Function marked as `__init` and can only be used during system boot.

Returns error code from `fb_find_mode` (see `fb_find_mode` function).

## Frame Buffer Fonts

Refer to the file `lib/fonts/fonts.c` for more information.

---

# Chapter 9. Input Subsystem

## Input core

## Name

struct input\_value — input value representation

## Synopsis

```
struct input_value {  
    __u16 type;  
    __u16 code;  
    __s32 value;  
};
```

## Members

type	type of value (EV_KEY, EV_ABS, etc)
code	the value code
value	the value

## Name

struct input\_dev — represents an input device

## Synopsis

```
struct input_dev {
    const char * name;
    const char * phys;
    const char * uniq;
    struct input_id id;
    unsigned long propbit[BITS_TO_LONGS(INPUT_PROP_CNT)];
    unsigned long evbit[BITS_TO_LONGS(EV_CNT)];
    unsigned long keybit[BITS_TO_LONGS(KEY_CNT)];
    unsigned long relbit[BITS_TO_LONGS(REL_CNT)];
    unsigned long absbit[BITS_TO_LONGS(ABS_CNT)];
    unsigned long mscbit[BITS_TO_LONGS(MSC_CNT)];
    unsigned long ledbit[BITS_TO_LONGS(LED_CNT)];
    unsigned long sndbit[BITS_TO_LONGS(SND_CNT)];
    unsigned long ffbit[BITS_TO_LONGS(FF_CNT)];
    unsigned long swbit[BITS_TO_LONGS(SW_CNT)];
    unsigned int hint_events_per_packet;
    unsigned int keycodemax;
    unsigned int keycodesize;
    void * keycode;
    int (* setkeycode) (struct input_dev *dev, const struct input_keymap_entry *ke, unsigned int code);
    int (* getkeycode) (struct input_dev *dev, struct input_keymap_entry *ke);
    struct ff_device * ff;
    unsigned int repeat_key;
    struct timer_list timer;
    int rep[REP_CNT];
    struct input_mt * mt;
    struct input_absinfo * absinfo;
    unsigned long key[BITS_TO_LONGS(KEY_CNT)];
    unsigned long led[BITS_TO_LONGS(LED_CNT)];
    unsigned long snd[BITS_TO_LONGS(SND_CNT)];
    unsigned long sw[BITS_TO_LONGS(SW_CNT)];
    int (* open) (struct input_dev *dev);
    void (* close) (struct input_dev *dev);
    int (* flush) (struct input_dev *dev, struct file *file);
    int (* event) (struct input_dev *dev, unsigned int type, unsigned int code, int value);
    struct input_handle __rcu * grab;
    spinlock_t event_lock;
    struct mutex mutex;
    unsigned int users;
    bool going_away;
    struct device dev;
    struct list_head h_list;
    struct list_head node;
    unsigned int num_vals;
    unsigned int max_vals;
    struct input_value * vals;
    bool devres_managed;
```

```
};
```

## Members

name	name of the device
phys	physical path to the device in the system hierarchy
uniq	unique identification code for the device (if device has it)
id	id of the device (struct input_id)
propbit[BITS_TO_LONGS(INPUT_PROP_CNT)]	bitmap of device properties and quirks
evbit[BITS_TO_LONGS(EV_CNT)]	bitmap of types of events supported by the device (EV_KEY, EV_REL, etc.)
keybit[BITS_TO_LONGS(KEY_CNT)]	bitmap of keys/buttons this device has
relbit[BITS_TO_LONGS(REL_CNT)]	bitmap of relative axes for the device
absbit[BITS_TO_LONGS(ABS_CNT)]	bitmap of absolute axes for the device
mscbit[BITS_TO_LONGS(MSC_CNT)]	map of miscellaneous events supported by the device
ledbit[BITS_TO_LONGS(LED_CNT)]	bitmap of leds present on the device
sndbit[BITS_TO_LONGS(SND_CNT)]	bitmap of sound effects supported by the device
ffbit[BITS_TO_LONGS(FF_CNT)]	bitmap of force feedback effects supported by the device
swbit[BITS_TO_LONGS(SW_CNT)]	bitmap of switches present on the device
hint_events_per_packet	average number of events generated by the device in a packet (between EV_SYN/SYN_REPORT events). Used by event handlers to estimate size of the buffer needed to hold events.
keycodemax	size of keycode table
keycodesize	size of elements in keycode table
keycode	map of scancodes to keycodes for this device
setkeycode	optional method to alter current keymap, used to implement sparse keymaps. If not supplied default mechanism will be used. The method is being called while holding event_lock and thus must not sleep
getkeycode	optional legacy method to retrieve current keymap.
ff	force feedback structure associated with the device if device supports force feedback effects
repeat_key	stores key code of the last key pressed; used to implement software autorepeat
timer	timer for software autorepeat



rep[REP_CNT]	current values for autorepeat parameters (delay, rate)
mt	pointer to multitouch state
absinfo	array of struct <code>input_absinfo</code> elements holding information about absolute axes (current value, min, max, flat, fuzz, resolution)
key[BITS_TO_LONGS(KEY_CNT)]	reflects current state of device's keys/buttons
led[BITS_TO_LONGS(LED_CNT)]	reflects current state of device's LEDs
snd[BITS_TO_LONGS(SND_CNT)]	reflects current state of sound effects
sw[BITS_TO_LONGS(SW_CNT)]	reflects current state of device's switches
open	this method is called when the very first user calls <code>input_open_device</code> . The driver must prepare the device to start generating events (start polling thread, request an IRQ, submit URB, etc.)
close	this method is called when the very last user calls <code>input_close_device</code> .
flush	purges the device. Most commonly used to get rid of force feedback effects loaded into the device when disconnecting from it
event	event handler for events sent <code>_to_</code> the device, like <code>EV_LED</code> or <code>EV_SND</code> . The device is expected to carry out the requested action (turn on a LED, play sound, etc.) The call is protected by <code>event_lock</code> and must not sleep
grab	input handle that currently has the device grabbed (via <code>EVIOCGRAB</code> ioctl). When a handle grabs a device it becomes sole recipient for all input events coming from the device
event_lock	this spinlock is is taken when input core receives and processes a new event for the device (in <code>input_event</code> ). Code that accesses and/or modifies parameters of a device (such as keymap or absmin, absmax, absfuzz, etc.) after device has been registered with input core must take this lock.
mutex	serializes calls to <code>open</code> , <code>close</code> and <code>flush</code> methods
users	stores number of users (input handlers) that opened this device. It is used by <code>input_open_device</code> and <code>input_close_device</code> to make sure that <code>dev-&gt;open</code> is only called when the first user opens device and <code>dev-&gt;close</code> is called when the very last user closes the device
going_away	marks devices that are in a middle of unregistering and causes <code>input_open_device*()</code> fail with <code>-ENODEV</code> .
dev	driver model's view of this device
h_list	list of input handles associated with the device. When accessing the list <code>dev-&gt;mutex</code> must be held

node	used to place the device onto input_dev_list
num_vals	number of values queued in the current frame
max_vals	maximum number of values queued in a frame
vals	array of values queued in the current frame
devres_managed	indicates that devices is managed with devres framework and needs not be explicitly unregistered or freed.

## Name

struct input\_handler — implements one of interfaces for input devices

## Synopsis

```
struct input_handler {
    void * private;
    void (* event) (struct input_handle *handle, unsigned int type, unsigned int code);
    void (* events) (struct input_handle *handle, const struct input_value *vals, unsigned int count);
    bool (* filter) (struct input_handle *handle, unsigned int type, unsigned int code);
    bool (* match) (struct input_handler *handler, struct input_dev *dev);
    int (* connect) (struct input_handler *handler, struct input_dev *dev, const struct input_device_id *id);
    void (* disconnect) (struct input_handle *handle);
    void (* start) (struct input_handle *handle);
    bool legacy_minors;
    int minor;
    const char * name;
    const struct input_device_id * id_table;
    struct list_head h_list;
    struct list_head node;
};
```

## Members

private	driver-specific data
event	event handler. This method is being called by input core with interrupts disabled and dev->event_lock spinlock held and so it may not sleep
events	event sequence handler. This method is being called by input core with interrupts disabled and dev->event_lock spinlock held and so it may not sleep
filter	similar to <i>event</i> ; separates normal event handlers from “filters”.
match	called after comparing device's id with handler's id_table to perform fine-grained matching between device and handler
connect	called when attaching a handler to an input device
disconnect	disconnects a handler from input device
start	starts handler for given handle. This function is called by input core right after connect method and also when a process that “grabbed” a device releases it
legacy_minors	set to true by drivers using legacy minor ranges
minor	beginning of range of 32 legacy minors for devices this driver can provide
name	name of the handler, to be shown in /proc/bus/input/handlers
id_table	pointer to a table of input_device_ids this driver can handle
h_list	list of input handles associated with the handler
node	for placing the driver onto input_handler_list

## Description

Input handlers attach to input devices and create input handles. There are likely several handlers attached to any given input device at the same time. All of them will get their copy of input event generated by the device.

The very same structure is used to implement input filters. Input core allows filters to run first and will not pass event to regular handlers if any of the filters indicate that the event should be filtered (by returning `true` from their `filter` method).

Note that input core serializes calls to `connect` and `disconnect` methods.

## Name

struct input\_handle — links input device with an input handler

## Synopsis

```
struct input_handle {  
    void * private;  
    int open;  
    const char * name;  
    struct input_dev * dev;  
    struct input_handler * handler;  
    struct list_head d_node;  
    struct list_head h_node;  
};
```

## Members

private	handler-specific data
open	counter showing whether the handle is 'open', i.e. should deliver events from its device
name	name given to the handle by handler that created it
dev	input device the handle is attached to
handler	handler that works with the device through this handle
d_node	used to put the handle on device's list of attached handles
h_node	used to put the handle on handler's list of handles from which it gets events

## Name

`input_set_events_per_packet` — tell handlers about the driver event rate

## Synopsis

```
void input_set_events_per_packet (struct input_dev * dev, int n_events);
```

## Arguments

*dev*            the input device used by the driver

*n\_events*    the average number of events between calls to `input_sync`

## Description

If the event rate sent from a device is unusually large, use this function to set the expected event rate. This will allow handlers to set up an appropriate buffer size for the event stream, in order to minimize information loss.

## Name

struct ff\_device — force-feedback part of an input device

## Synopsis

```
struct ff_device {
    int (* upload) (struct input_dev *dev, struct ff_effect *effect, struct ff_effect *effect);
    int (* erase) (struct input_dev *dev, int effect_id);
    int (* playback) (struct input_dev *dev, int effect_id, int value);
    void (* set_gain) (struct input_dev *dev, u16 gain);
    void (* set_autocenter) (struct input_dev *dev, u16 magnitude);
    void (* destroy) (struct ff_device *);
    void * private;
    unsigned long ffbits[BITS_TO_LONGS(FF_CNT)];
    struct mutex mutex;
    int max_effects;
    struct ff_effect * effects;
    struct file * effect_owners[];
};
```

## Members

upload	Called to upload an new effect into device
erase	Called to erase an effect from device
playback	Called to request device to start playing specified effect
set_gain	Called to set specified gain
set_autocenter	Called to auto-center device
destroy	called by input core when parent input device is being destroyed
private	driver-specific data, will be freed automatically
ffbits[BITS_TO_LONGS(FF_CNT)]	bitmap of force feedback capabilities truly supported by device (not emulated like ones in input_dev->ffbits)
mutex	mutex for serializing access to the device
max_effects	maximum number of effects supported by device
effects	pointer to an array of effects currently loaded into device
effect_owners[]	array of effect owners; when file handle owning an effect gets closed the effect is automatically erased

## Description

Every force-feedback device must implement upload and playback methods; erase is optional. set\_gain and set\_autocenter need only be implemented if driver sets up FF\_GAIN and FF\_AUTOCENTER bits.

Note that `playback`, `set_gain` and `set_autocenter` are called with `dev->event_lock` spinlock held and interrupts off and thus may not sleep.



## Name

`input_event` — report new input event

## Synopsis

```
void input_event (struct input_dev * dev, unsigned int type, unsigned  
int code, int value);
```

## Arguments

*dev*      device that generated the event

*type*     type of the event

*code*     event code

*value*    value of the event

## Description

This function should be used by drivers implementing various input devices to report input events. See also `input_inject_event`.

## NOTE

`input_event` may be safely used right after input device was allocated with `input_allocate_device`, even before it is registered with `input_register_device`, but the event will not reach any of the input handlers. Such early invocation of `input_event` may be used to 'seed' initial state of a switch or initial position of absolute axis, etc.

## Name

`input_inject_event` — send input event from input handler

## Synopsis

```
void input_inject_event (struct input_handle * handle, unsigned int  
type, unsigned int code, int value);
```

## Arguments

*handle*    input handle to send event through

*type*      type of the event

*code*      event code

*value*     value of the event

## Description

Similar to `input_event` but will ignore event if device is “grabbed” and handle injecting event is not the one that owns the device.

## Name

`input_alloc_absinfo` — allocates array of `input_absinfo` structs

## Synopsis

```
void input_alloc_absinfo (struct input_dev * dev);
```

## Arguments

*dev* the input device emitting absolute events

## Description

If the `absinfo` struct the caller asked for is already allocated, this functions will not do anything.

## Name

`input_grab_device` — grabs device for exclusive use

## Synopsis

```
int input_grab_device (struct input_handle * handle);
```

## Arguments

*handle*    input handle that wants to own the device

## Description

When a device is grabbed by an input handle all events generated by the device are delivered only to this handle. Also events injected by other input handles are ignored while device is grabbed.

## Name

`input_release_device` — release previously grabbed device

## Synopsis

```
void input_release_device (struct input_handle * handle);
```

## Arguments

*handle*   input handle that owns the device

## Description

Releases previously grabbed device so that other input handles can start receiving input events. Upon release all handlers attached to the device have their `start` method called so they have a change to synchronize device state with the rest of the system.

## Name

`input_open_device` — open input device

## Synopsis

```
int input_open_device (struct input_handle * handle);
```

## Arguments

*handle*    handle through which device is being accessed

## Description

This function should be called by input handlers when they want to start receive events from given input device.

## Name

`input_close_device` — close input device

## Synopsis

```
void input_close_device (struct input_handle * handle);
```

## Arguments

*handle*    handle through which device is being accessed

## Description

This function should be called by input handlers when they want to stop receive events from given input device.

## Name

`input_scancode_to_scalar` — converts scancode in struct `input_keymap_entry`

## Synopsis

```
int input_scancode_to_scalar (const struct input_keymap_entry * ke,  
unsigned int * scancode);
```

## Arguments

*ke*            keymap entry containing scancode to be converted.

*scancode*    pointer to the location where converted scancode should be stored.

## Description

This function is used to convert scancode stored in struct `keymap_entry` into scalar form understood by legacy keymap handling methods. These methods expect scancodes to be represented as 'unsigned int'.



## Name

`input_get_keycode` — retrieve keycode currently mapped to a given scancode

## Synopsis

```
int input_get_keycode (struct input_dev * dev, struct input_keymap_entry  
* ke);
```

## Arguments

*dev*    input device which keymap is being queried

*ke*     keymap entry

## Description

This function should be called by anyone interested in retrieving current keymap. Presently evdev handlers use it.

## Name

`input_set_keycode` — attribute a keycode to a given scancode

## Synopsis

```
int  input_set_keycode (struct  input_dev  *  dev,  const  struct
input_keymap_entry *  ke);
```

## Arguments

*dev* input device which keymap is being updated

*ke* new keymap entry

## Description

This function should be called by anyone needing to update current keymap. Presently keyboard and evdev handlers use it.

## Name

`input_reset_device` — reset/restore the state of input device

## Synopsis

```
void input_reset_device (struct input_dev * dev);
```

## Arguments

*dev*   input device whose state needs to be reset

## Description

This function tries to reset the state of an opened input device and bring internal state and state if the hardware in sync with each other. We mark all keys as released, restore LED state, repeat rate, etc.

## Name

`input_allocate_device` — allocate memory for new input device

## Synopsis

```
struct input_dev * input_allocate_device ( void );
```

## Arguments

*void* no arguments

## Description

Returns prepared struct `input_dev` or `NULL`.

## NOTE

Use `input_free_device` to free devices that have not been registered; `input_unregister_device` should be used for already registered devices.

## Name

`devm_input_allocate_device` — allocate managed input device

## Synopsis

```
struct input_dev * devm_input_allocate_device (struct device * dev);
```

## Arguments

*dev*    device owning the input device being created

## Description

Returns prepared struct `input_dev` or `NULL`.

Managed input devices do not need to be explicitly unregistered or freed as it will be done automatically when owner device unbinds from its driver (or binding fails). Once managed input device is allocated, it is ready to be set up and registered in the same fashion as regular input device. There are no special `devm_input_device_[un]register` variants, regular ones work with both managed and unmanaged devices, should you need them. In most cases however, managed input device need not be explicitly unregistered or freed.

## NOTE

the owner device is set up as parent of input device and users should not override it.

## Name

`input_free_device` — free memory occupied by `input_dev` structure

## Synopsis

```
void input_free_device (struct input_dev * dev);
```

## Arguments

*dev*   input device to free

## Description

This function should only be used if `input_register_device` was not called yet or if it failed. Once device was registered use `input_unregister_device` and memory will be freed once last reference to the device is dropped.

Device should be allocated by `input_allocate_device`.

## NOTE

If there are references to the input device then memory will not be freed until last reference is dropped.

## Name

`input_set_capability` — mark device as capable of a certain event

## Synopsis

```
void input_set_capability (struct input_dev * dev, unsigned int type,  
unsigned int code);
```

## Arguments

*dev* device that is capable of emitting or accepting event

*type* type of the event (EV\_KEY, EV\_REL, etc...)

*code* event code

## Description

In addition to setting up corresponding bit in appropriate capability bitmap the function also adjusts `dev->evbit`.

## Name

`input_enable_softrepeat` — enable software autorepeat

## Synopsis

```
void input_enable_softrepeat (struct input_dev * dev, int delay, int  
period);
```

## Arguments

*dev*        input device

*delay*     repeat delay

*period*    repeat period

## Description

Enable software autorepeat on the input device.



## Name

`input_register_device` — register device with input core

## Synopsis

```
int input_register_device (struct input_dev * dev);
```

## Arguments

*dev*    device to be registered

## Description

This function registers device with input core. The device must be allocated with `input_allocate_device` and all its capabilities set up before registering. If function fails the device must be freed with `input_free_device`. Once device has been successfully registered it can be unregistered with `input_unregister_device`; `input_free_device` should not be called in this case.

Note that this function is also used to register managed input devices (ones allocated with `devm_input_allocate_device`). Such managed input devices need not be explicitly unregistered or freed, their tear down is controlled by the devres infrastructure. It is also worth noting that tear down of managed input devices is internally a 2-step process: registered managed input device is first unregistered, but stays in memory and can still handle `input_event` calls (although events will not be delivered anywhere). The freeing of managed input device will happen later, when devres stack is unwound to the point where device allocation was made.

## Name

`input_unregister_device` — unregister previously registered device

## Synopsis

```
void input_unregister_device (struct input_dev * dev);
```

## Arguments

*dev*    device to be unregistered

## Description

This function unregisters an input device. Once device is unregistered the caller should not try to access it as it may get freed at any moment.

## Name

`input_register_handler` — register a new input handler

## Synopsis

```
int input_register_handler (struct input_handler * handler);
```

## Arguments

*handler* handler to be registered

## Description

This function registers a new input handler (interface) for input devices in the system and attaches it to all input devices that are compatible with the handler.

## Name

`input_unregister_handler` — unregisters an input handler

## Synopsis

```
void input_unregister_handler (struct input_handler * handler);
```

## Arguments

*handler* handler to be unregistered

## Description

This function disconnects a handler from its input devices and removes it from lists of known handlers.

## Name

`input_handler_for_each_handle` — handle iterator

## Synopsis

```
int input_handler_for_each_handle (struct input_handler * handler, void  
* data, int (*fn) (struct input_handle *, void *));
```

## Arguments

*handler*    input handler to iterate

*data*        data for the callback

*fn*          function to be called for each handle

## Description

Iterate over *bus*'s list of devices, and call *fn* for each, passing it *data* and stop when *fn* returns a non-zero value. The function is using RCU to traverse the list and therefore may be using in atomic contexts. The *fn* callback is invoked from RCU critical section and thus must not sleep.

## Name

`input_register_handle` — register a new input handle

## Synopsis

```
int input_register_handle (struct input_handle * handle);
```

## Arguments

*handle*    handle to register

## Description

This function puts a new input handle onto device's and handler's lists so that events can flow through it once it is opened using `input_open_device`.

This function is supposed to be called from handler's `connect` method.

## Name

`input_unregister_handle` — unregister an input handle

## Synopsis

```
void input_unregister_handle (struct input_handle * handle);
```

## Arguments

*handle*    handle to unregister

## Description

This function removes input handle from device's and handler's lists.

This function is supposed to be called from handler's `disconnect` method.

## Name

`input_get_new_minor` — allocates a new input minor number

## Synopsis

```
int input_get_new_minor (int legacy_base, unsigned int legacy_num, bool  
allow_dynamic);
```

## Arguments

*legacy\_base*      beginning of the legacy range to be searched

*legacy\_num*      size of legacy range

*allow\_dynamic*   whether we can also take ID from the dynamic range

## Description

This function allocates a new device minor for from input major namespace. Caller can request legacy minor by specifying *legacy\_base* and *legacy\_num* parameters and whether ID can be allocated from dynamic range if there are no free IDs in legacy range.



## Name

`input_free_minor` — release previously allocated minor

## Synopsis

```
void input_free_minor (unsigned int minor);
```

## Arguments

*minor*    minor to be released

## Description

This function releases previously allocated input minor so that it can be reused later.

## Name

`input_ff_upload` — upload effect into force-feedback device

## Synopsis

```
int input_ff_upload (struct input_dev * dev, struct ff_effect * effect,  
struct file * file);
```

## Arguments

<i>dev</i>	input device
<i>effect</i>	effect to be uploaded
<i>file</i>	owner of the effect

## Name

`input_ff_erase` — erase a force-feedback effect from device

## Synopsis

```
int input_ff_erase (struct input_dev * dev, int effect_id, struct file  
* file);
```

## Arguments

*dev*            input device to erase effect from

*effect\_id*    id of the effect to be erased

*file*            purported owner of the request

## Description

This function erases a force-feedback effect from specified device. The effect will only be erased if it was uploaded through the same file handle that is requesting erase.

## Name

`input_ff_event` — generic handler for force-feedback events

## Synopsis

```
int input_ff_event (struct input_dev * dev, unsigned int type, unsigned  
int code, int value);
```

## Arguments

<i>dev</i>	input device to send the effect to
<i>type</i>	event type (anything but EV_FF is ignored)
<i>code</i>	event code
<i>value</i>	event value

## Name

`input_ff_create` — create force-feedback device

## Synopsis

```
int input_ff_create (struct input_dev * dev, unsigned int max_effects);
```

## Arguments

*dev*                   input device supporting force-feedback

*max\_effects*   maximum number of effects supported by the device

## Description

This function allocates all necessary memory for a force feedback portion of an input device and installs all default handlers. *dev*->ffbit should be already set up before calling this function. Once ff device is created you need to setup its upload, erase, playback and other handlers before registering input device

## Name

`input_ff_destroy` — frees force feedback portion of input device

## Synopsis

```
void input_ff_destroy (struct input_dev * dev);
```

## Arguments

*dev* input device supporting force feedback

## Description

This function is only needed in error path as input core will automatically free force feedback structures when device is destroyed.

## Name

`input_ff_create_memless` — create memoryless force-feedback device

## Synopsis

```
int input_ff_create_memless (struct input_dev * dev, void * data, int
(*play_effect) (struct input_dev *, void *, struct ff_effect *));
```

## Arguments

<i>dev</i>	input device supporting force-feedback
<i>data</i>	driver-specific data to be passed into <i>play_effect</i>
<i>play_effect</i>	driver-specific method for playing FF effect

## Multitouch Library

## Name

struct input\_mt\_slot — represents the state of an input MT slot

## Synopsis

```
struct input_mt_slot {  
    int abs[ABS_MT_LAST - ABS_MT_FIRST + 1];  
    unsigned int frame;  
    unsigned int key;  
};
```

## Members

abs[ABS_MT_LAST - ABS_MT_FIRST + 1]	holds current values of ABS_MT axes for this slot
frame	last frame at which input_mt_report_slot_state was called
key	optional driver designation of this slot



## Name

struct input\_mt — state of tracked contacts

## Synopsis

```
struct input_mt {  
    int trkid;  
    int num_slots;  
    int slot;  
    unsigned int flags;  
    unsigned int frame;  
    int * red;  
    struct input_mt_slot slots[];  
};
```

## Members

trkid	stores MT tracking ID for the next contact
num_slots	number of MT slots the device uses
slot	MT slot currently being transmitted
flags	input_mt operation flags
frame	increases every time <code>input_mt_sync_frame</code> is called
red	reduced cost matrix for in-kernel tracking
slots[]	array of slots holding current values of tracked contacts

## Name

struct input\_mt\_pos — contact position

## Synopsis

```
struct input_mt_pos {  
    s16 x;  
    s16 y;  
};
```

## Members

x   horizontal coordinate

y   vertical coordinate

## Name

`input_mt_init_slots` — initialize MT input slots

## Synopsis

```
int input_mt_init_slots (struct input_dev * dev, unsigned int num_slots,  
unsigned int flags);
```

## Arguments

*dev*           input device supporting MT events and finger tracking

*num\_slots*    number of slots used by the device

*flags*         mt tasks to handle in core

## Description

This function allocates all necessary memory for MT slot handling in the input device, prepares the `ABS_MT_SLOT` and `ABS_MT_TRACKING_ID` events for use and sets up appropriate buffers. Depending on the flags set, it also performs pointer emulation and frame synchronization.

May be called repeatedly. Returns `-EINVAL` if attempting to reinitialize with a different number of slots.

## Name

`input_mt_destroy_slots` — frees the MT slots of the input device

## Synopsis

```
void input_mt_destroy_slots (struct input_dev * dev);
```

## Arguments

*dev* input device with allocated MT slots

## Description

This function is only needed in error path as the input core will automatically free the MT slots when the device is destroyed.

## Name

`input_mt_report_slot_state` — report contact state

## Synopsis

```
void input_mt_report_slot_state (struct input_dev * dev, unsigned int  
tool_type, bool active);
```

## Arguments

<i>dev</i>	input device with allocated MT slots
<i>tool_type</i>	the tool type to use in this slot
<i>active</i>	true if contact is active, false otherwise

## Description

Reports a contact via `ABS_MT_TRACKING_ID`, and optionally `ABS_MT_TOOL_TYPE`. If `active` is true and the slot is currently inactive, or if the tool type is changed, a new tracking id is assigned to the slot. The tool type is only reported if the corresponding `absbit` field is set.

## Name

`input_mt_report_finger_count` — report contact count

## Synopsis

```
void input_mt_report_finger_count (struct input_dev * dev, int count);
```

## Arguments

*dev*     input device with allocated MT slots

*count*   the number of contacts

## Description

Reports the contact count via `BTN_TOOL_FINGER`, `BTN_TOOL_DOUBLETAP`, `BTN_TOOL_TRIPLETAP` and `BTN_TOOL_QUADTAP`.

The input core ensures only the KEY events already setup for this device will produce output.

## Name

`input_mt_report_pointer_emulation` — common pointer emulation

## Synopsis

```
void input_mt_report_pointer_emulation (struct input_dev * dev, bool
use_count);
```

## Arguments

*dev*           input device with allocated MT slots

*use\_count*   report number of active contacts as finger count

## Description

Performs legacy pointer emulation via BTN\_TOUCH, ABS\_X, ABS\_Y and ABS\_PRESSURE. Touchpad finger count is emulated if *use\_count* is true.

The input core ensures only the KEY and ABS axes already setup for this device will produce output.

## Name

`input_mt_drop_unused` — Inactivate slots not seen in this frame

## Synopsis

```
void input_mt_drop_unused (struct input_dev * dev);
```

## Arguments

*dev* input device with allocated MT slots

## Description

Lift all slots not seen since the last call to this function.



## Name

`input_mt_sync_frame` — synchronize mt frame

## Synopsis

```
void input_mt_sync_frame (struct input_dev * dev);
```

## Arguments

*dev* input device with allocated MT slots

## Description

Close the frame and prepare the internal state for a new one. Depending on the flags, marks unused slots as inactive and performs pointer emulation.

## Name

`input_mt_assign_slots` — perform a best-match assignment

## Synopsis

```
int input_mt_assign_slots (struct input_dev * dev, int * slots, const
struct input_mt_pos * pos, int num_pos, int dmax);
```

## Arguments

<i>dev</i>	input device with allocated MT slots
<i>slots</i>	the slot assignment to be filled
<i>pos</i>	the position array to match
<i>num_pos</i>	number of positions
<i>dmax</i>	maximum ABS_MT_POSITION displacement (zero for infinite)

## Description

Performs a best match against the current contacts and returns the slot assignment list. New contacts are assigned to unused slots.

The assignments are balanced so that all coordinate displacements are below the euclidian distance `dmax`. If no such assignment can be found, some contacts are assigned to unused slots.

Returns zero on success, or negative error in case of failure.

## Name

`input_mt_get_slot_by_key` — return slot matching key

## Synopsis

```
int input_mt_get_slot_by_key (struct input_dev * dev, int key);
```

## Arguments

*dev* input device with allocated MT slots

*key* the key of the sought slot

## Description

Returns the slot of the given key, if it exists, otherwise set the key on the first unused slot and return.

If no available slot can be found, -1 is returned. Note that for this function to work properly, `input_mt_sync_frame` has to be called at each frame.

## Polled input devices

## Name

struct input\_polled\_dev — simple polled input device

## Synopsis

```
struct input_polled_dev {
    void * private;
    void (* open) (struct input_polled_dev *dev);
    void (* close) (struct input_polled_dev *dev);
    void (* poll) (struct input_polled_dev *dev);
    unsigned int poll_interval;
    unsigned int poll_interval_max;
    unsigned int poll_interval_min;
    struct input_dev * input;
};
```

## Members

private	private driver data.
open	driver-supplied method that prepares device for polling (enabled the device and maybe flushes device state).
close	driver-supplied method that is called when device is no longer being polled. Used to put device into low power mode.
poll	driver-supplied method that polls the device and posts input events (mandatory).
poll_interval	specifies how often the <code>poll</code> method should be called. Defaults to 500 msec unless overridden when registering the device.
poll_interval_max	specifies upper bound for the poll interval. Defaults to the initial value of <i>poll_interval</i> .
poll_interval_min	specifies lower bound for the poll interval. Defaults to 0.
input	input device structure associated with the polled device. Must be properly initialized by the driver (id, name, phys, bits).

## Description

Polled input device provides a skeleton for supporting simple input devices that do not raise interrupts but have to be periodically scanned or polled to detect changes in their state.

## Name

`input_allocate_polled_device` — allocate memory for polled device

## Synopsis

```
struct input_polled_dev * input_allocate_polled_device ( void );
```

## Arguments

*void* no arguments

## Description

The function allocates memory for a polled device and also for an input device associated with this polled device.

## Name

`devm_input_allocate_polled_device` — allocate managed polled device

## Synopsis

```
struct input_polled_dev * devm_input_allocate_polled_device (struct
device * dev);
```

## Arguments

*dev* device owning the polled device being created

## Description

Returns prepared struct `input_polled_dev` or `NULL`.

Managed polled input devices do not need to be explicitly unregistered or freed as it will be done automatically when owner device unbinds from \* its driver (or binding fails). Once such managed polled device is allocated, it is ready to be set up and registered in the same fashion as regular polled input devices (using `input_register_polled_device` function).

If you want to manually unregister and free such managed polled devices, it can be still done by calling `input_unregister_polled_device` and `input_free_polled_device`, although it is rarely needed.

## NOTE

the owner device is set up as parent of input device and users should not override it.

## Name

`input_free_polled_device` — free memory allocated for polled device

## Synopsis

```
void input_free_polled_device (struct input_polled_dev * dev);
```

## Arguments

*dev*    device to free

## Description

The function frees memory allocated for polling device and drops reference to the associated input device.

## Name

`input_register_polled_device` — register polled device

## Synopsis

```
int input_register_polled_device (struct input_polled_dev * dev);
```

## Arguments

*dev* device to register

## Description

The function registers previously initialized polled input device with input layer. The device should be allocated with call to `input_allocate_polled_device`. Callers should also set up `poll` method and set up capabilities (id, name, phys, bits) of the corresponding `input_dev` structure.



## Name

`input_unregister_polled_device` — unregister polled device

## Synopsis

```
void input_unregister_polled_device (struct input_polled_dev * dev);
```

## Arguments

*dev* device to unregister

## Description

The function unregisters previously registered polled input device from input layer. Polling is stopped and device is ready to be freed with call to `input_free_polled_device`.

## Matrix keyboards/keypads

## Name

struct matrix\_keymap\_data — keymap for matrix keyboards

## Synopsis

```
struct matrix_keymap_data {  
    const uint32_t * keymap;  
    unsigned int keymap_size;  
};
```

## Members

keymap	pointer to array of uint32 values encoded with KEY macro representing keymap
keymap_size	number of entries (initialized) in this keymap

## Description

This structure is supposed to be used by platform code to supply keymaps to drivers that implement matrix-like keypads/keyboards.

## Name

struct matrix\_keypad\_platform\_data — platform-dependent keypad data

## Synopsis

```
struct matrix_keypad_platform_data {
    const struct matrix_keymap_data * keymap_data;
    const unsigned int * row_gpios;
    const unsigned int * col_gpios;
    unsigned int num_row_gpios;
    unsigned int num_col_gpios;
    unsigned int col_scan_delay_us;
    unsigned int debounce_ms;
    unsigned int clustered_irq;
    unsigned int clustered_irq_flags;
    bool active_low;
    bool wakeup;
    bool no_autorepeat;
};
```

## Members

keymap_data	pointer to matrix_keymap_data
row_gpios	pointer to array of gpio numbers representing rows
col_gpios	pointer to array of gpio numbers representing columns
num_row_gpios	actual number of row gpios used by device
num_col_gpios	actual number of col gpios used by device
col_scan_delay_us	delay, measured in microseconds, that is needed before we can keypad after activating column gpio
debounce_ms	debounce interval in milliseconds
clustered_irq	may be specified if interrupts of all row/column GPIOs are bundled to one single irq
clustered_irq_flags	flags that are needed for the clustered irq
active_low	gpio polarity
wakeup	controls whether the device should be set up as wakeup source
no_autorepeat	disable key autorepeat

## Description

This structure represents platform-specific data that is used by matrix\_keypad driver to perform proper initialization.

## Name

`matrix_keypad_parse_of_params` — Read parameters from matrix-keypad node

## Synopsis

```
int matrix_keypad_parse_of_params (struct device * dev, unsigned int *  
rows, unsigned int * cols);
```

## Arguments

*dev*    Device containing of\_node

*rows*   Returns number of matrix rows

*cols*   Returns number of matrix columns *return* 0 if OK, <0 on error

## Sparse keymap support

## Name

struct key\_entry — keymap entry for use in sparse keymap

## Synopsis

```
struct key_entry {  
    int type;  
    u32 code;  
    union {unnamed_union};  
};
```

## Members

type	Type of the key entry (KE_KEY, KE_SW, KE_VSW, KE_END); drivers are allowed to extend the list with their own private definitions.
code	Device-specific data identifying the button/switch
{unnamed_union}	anonymous

## Description

This structure defines an entry in a sparse keymap used by some input devices for which traditional table-based approach is not suitable.

## Name

`sparse_keymap_entry_from_scancode` — perform sparse keymap lookup

## Synopsis

```
struct key_entry * sparse_keymap_entry_from_scancode (struct input_dev  
* dev, unsigned int code);
```

## Arguments

*dev*    Input device using sparse keymap

*code*   Scan code

## Description

This function is used to perform struct `key_entry` lookup in an input device using sparse keymap.

## Name

`sparse_keymap_entry_from_keycode` — perform sparse keymap lookup

## Synopsis

```
struct key_entry * sparse_keymap_entry_from_keycode (struct input_dev  
* dev, unsigned int keycode);
```

## Arguments

*dev*        Input device using sparse keymap

*keycode*   Key code

## Description

This function is used to perform struct `key_entry` lookup in an input device using sparse keymap.

## Name

`sparse_keymap_setup` — set up sparse keymap for an input device

## Synopsis

```
int sparse_keymap_setup (struct input_dev * dev, const struct key_entry  
* keymap, int (*setup) (struct input_dev *, struct key_entry *));
```

## Arguments

*dev*        Input device

*keymap*    Keymap in form of array of `key_entry` structures ending with `KE_END` type entry

*setup*     Function that can be used to adjust keymap entries depending on device's deeds, may be `NULL`

## Description

The function calculates size and allocates copy of the original keymap after which sets up input device event bits appropriately. Before destroying input device allocated keymap should be freed with a call to `sparse_keymap_free`.



## Name

`sparse_keymap_free` — free memory allocated for sparse keymap

## Synopsis

```
void sparse_keymap_free (struct input_dev * dev);
```

## Arguments

*dev*    Input device using sparse keymap

## Description

This function is used to free memory allocated by sparse keymap in an input device that was set up by `sparse_keymap_setup`.

## NOTE

It is safe to call this function while input device is still registered (however the drivers should care not to try to use freed keymap and thus have to shut off interrupts/polling before freeing the keymap).

## Name

`sparse_keymap_report_entry` — report event corresponding to given key entry

## Synopsis

```
void sparse_keymap_report_entry (struct input_dev * dev, const struct  
key_entry * ke, unsigned int value, bool autorelease);
```

## Arguments

<i>dev</i>	Input device for which event should be reported
<i>ke</i>	key entry describing event
<i>value</i>	Value that should be reported (ignored by KE_SW entries)
<i>autorelease</i>	Signals whether release event should be emitted for KE_KEY entries right after reporting press event, ignored by all other entries

## Description

This function is used to report input event described by given struct `key_entry`.

## Name

`sparse_keymap_report_event` — report event corresponding to given scancode

## Synopsis

```
bool sparse_keymap_report_event (struct input_dev * dev, unsigned int
code, unsigned int value, bool autorelease);
```

## Arguments

<i>dev</i>	Input device using sparse keymap
<i>code</i>	Scan code
<i>value</i>	Value that should be reported (ignored by KE_SW entries)
<i>autorelease</i>	Signals whether release event should be emitted for KE_KEY entries right after reporting press event, ignored by all other entries

## Description

This function is used to perform lookup in an input device using sparse keymap and report corresponding event. Returns `true` if lookup was successful and `false` otherwise.

---

# Chapter 10. Serial Peripheral Interface (SPI)

SPI is the "Serial Peripheral Interface", widely used with embedded systems because it is a simple and efficient interface: basically a multiplexed shift register. Its three signal wires hold a clock (SCK, often in the range of 1-20 MHz), a "Master Out, Slave In" (MOSI) data line, and a "Master In, Slave Out" (MISO) data line. SPI is a full duplex protocol; for each bit shifted out the MOSI line (one per clock) another is shifted in on the MISO line. Those bits are assembled into words of various sizes on the way to and from system memory. An additional chipselect line is usually active-low (nCS); four signals are normally used for each peripheral, plus sometimes an interrupt.

The SPI bus facilities listed here provide a generalized interface to declare SPI busses and devices, manage them according to the standard Linux driver model, and perform input/output operations. At this time, only "master" side interfaces are supported, where Linux talks to SPI peripherals and does not implement such a peripheral itself. (Interfaces to support implementing SPI slaves would necessarily look different.)

The programming interface is structured around two kinds of driver, and two kinds of device. A "Controller Driver" abstracts the controller hardware, which may be as simple as a set of GPIO pins or as complex as a pair of FIFOs connected to dual DMA engines on the other side of the SPI shift register (maximizing throughput). Such drivers bridge between whatever bus they sit on (often the platform bus) and SPI, and expose the SPI side of their device as a struct `spi_master`. SPI devices are children of that master, represented as a struct `spi_device` and manufactured from struct `spi_board_info` descriptors which are usually provided by board-specific initialization code. A struct `spi_driver` is called a "Protocol Driver", and is bound to a `spi_device` using normal driver model calls.

The I/O model is a set of queued messages. Protocol drivers submit one or more struct `spi_message` objects, which are processed and completed asynchronously. (There are synchronous wrappers, however.) Messages are built from one or more struct `spi_transfer` objects, each of which wraps a full duplex SPI transfer. A variety of protocol tweaking options are needed, because different chips adopt very different policies for how they use the bits transferred with SPI.

## Name

struct spi\_statistics — statistics for spi transfers

## Synopsis

```
struct spi_statistics {
    spinlock_t lock;
    unsigned long messages;
    unsigned long transfers;
    unsigned long errors;
    unsigned long timedout;
    unsigned long spi_sync;
    unsigned long spi_sync_immediate;
    unsigned long spi_async;
    unsigned long long bytes;
    unsigned long long bytes_rx;
    unsigned long long bytes_tx;
#define SPI_STATISTICS_HISTO_SIZE 17
    unsigned long transfer_bytes_histo[SPI_STATISTICS_HISTO_SIZE];
};
```

## Members

lock	lock protecting this structure
messages	number of spi-messages handled
transfers	number of spi_transfers handled
errors	number of errors during spi_transfer
timedout	number of timeouts during spi_transfer
spi_sync	number of times spi_sync is used
spi_sync_immediate	number of times spi_sync is executed immediately in calling context without queuing and scheduling
spi_async	number of times spi_async is used
bytes	number of bytes transferred to/from device
bytes_rx	number of bytes received from device
bytes_tx	number of bytes sent to device
transfer_bytes_histo[SPI_STATISTICS_HISTO_SIZE]	transfer bytes histogram

## Name

struct spi\_device — Master side proxy for an SPI slave device

## Synopsis

```
struct spi_device {
    struct device dev;
    struct spi_master * master;
    u32 max_speed_hz;
    u8 chip_select;
    u8 bits_per_word;
    u16 mode;
#define SPI_CPHA 0x01
#define SPI_CPOL 0x02
#define SPI_MODE_0 (0|0)
#define SPI_MODE_1 (0|SPI_CPHA)
#define SPI_MODE_2 (SPI_CPOL|0)
#define SPI_MODE_3 (SPI_CPOL|SPI_CPHA)
#define SPI_CS_HIGH 0x04
#define SPI_LSB_FIRST 0x08
#define SPI_3WIRE 0x10
#define SPI_LOOP 0x20
#define SPI_NO_CS 0x40
#define SPI_READY 0x80
#define SPI_TX_DUAL 0x100
#define SPI_TX_QUAD 0x200
#define SPI_RX_DUAL 0x400
#define SPI_RX_QUAD 0x800
    int irq;
    void * controller_state;
    void * controller_data;
    char modalias[SPI_NAME_SIZE];
    int cs_gpio;
    struct spi_statistics statistics;
};
```

## Members

dev	Driver model representation of the device.
master	SPI controller used with the device.
max_speed_hz	Maximum clock rate to be used with this chip (on this board); may be changed by the device's driver. The spi_transfer.speed_hz can override this for each transfer.
chip_select	Chipselect, distinguishing chips handled by <i>master</i> .
bits_per_word	Data transfers involve one or more words; word sizes like eight or 12 bits are common. In-memory wordsizes are powers of two bytes (e.g. 20 bit samples use 32 bits). This may be changed by the device's driver, or left at the default (0) indicating protocol words are eight

	bit bytes. The <code>spi_transfer.bits_per_word</code> can override this for each transfer.
<code>mode</code>	The <code>spi</code> mode defines how data is clocked out and in. This may be changed by the device's driver. The “active low” default for chipselect mode can be overridden (by specifying <code>SPI_CS_HIGH</code> ) as can the “MSB first” default for each word in a transfer (by specifying <code>SPI_LSB_FIRST</code> ).
<code>irq</code>	Negative, or the number passed to <code>request_irq</code> to receive interrupts from this device.
<code>controller_state</code>	Controller's runtime state
<code>controller_data</code>	Board-specific definitions for controller, such as FIFO initialization parameters; from <code>board_info.controller_data</code>
<code>modalias[SPI_NAME_SIZE]</code>	Name of the driver to use with this device, or an alias for that name. This appears in the sysfs “modalias” attribute for driver coldplugging, and in uevents used for hotplugging
<code>cs_gpio</code>	gpio number of the chipselect line (optional, <code>-ENOENT</code> when not using a GPIO line)
<code>statistics</code>	statistics for the <code>spi_device</code>

## Description

A *spi\_device* is used to interchange data between an SPI slave (usually a discrete chip) and CPU memory.

In *dev*, the `platform_data` is used to hold information about this device that's meaningful to the device's protocol driver, but not to its controller. One example might be an identifier for a chip variant with slightly different functionality; another might be information about how this particular board wires the chip's pins.

## Name

struct spi\_driver — Host side “protocol” driver

## Synopsis

```
struct spi_driver {
    const struct spi_device_id * id_table;
    int (* probe) (struct spi_device *spi);
    int (* remove) (struct spi_device *spi);
    void (* shutdown) (struct spi_device *spi);
    struct device_driver driver;
};
```

## Members

id_table	List of SPI devices supported by this driver
probe	Binds this driver to the spi device. Drivers can verify that the device is actually present, and may need to configure characteristics (such as bits_per_word) which weren't needed for the initial configuration done during system setup.
remove	Unbinds this driver from the spi device
shutdown	Standard shutdown callback used during system state transitions such as powerdown/halt and kexec
driver	SPI device drivers should initialize the name and owner field of this structure.

## Description

This represents the kind of device driver that uses SPI messages to interact with the hardware at the other end of a SPI link. It's called a “protocol” driver because it works through messages rather than talking directly to SPI hardware (which is what the underlying SPI controller driver does to pass those messages). These protocols are defined in the specification for the device(s) supported by the driver.

As a rule, those device protocols represent the lowest level interface supported by a driver, and it will support upper level interfaces too. Examples of such upper levels include frameworks like MTD, networking, MMC, RTC, filesystem character device nodes, and hardware monitoring.



## Name

`spi_unregister_driver` — reverse effect of `spi_register_driver`

## Synopsis

```
void spi_unregister_driver (struct spi_driver * sdrv);
```

## Arguments

*sdrv* the driver to unregister

## Context

can sleep

## Name

`module_spi_driver` — Helper macro for registering a SPI driver

## Synopsis

```
module_spi_driver ( __spi_driver );
```

## Arguments

`__spi_driver` `spi_driver` struct

## Description

Helper macro for SPI drivers which do not do anything special in module init/exit. This eliminates a lot of boilerplate. Each module may only use this macro once, and calling it replaces `module_init` and `module_exit`

## Name

struct spi\_master — interface to SPI master controller

## Synopsis

```
struct spi_master {
    struct device dev;
    struct list_head list;
    s16 bus_num;
    u16 num_chipselect;
    u16 dma_alignment;
    u16 mode_bits;
    u32 bits_per_word_mask;
#define SPI_BPW_MASK(bits) BIT((bits) - 1)
#define SPI_BIT_MASK(bits) (((bits) == 32) ? ~0U : (BIT(bits) - 1))
#define SPI_BPW_RANGE_MASK(min# max) (SPI_BIT_MASK(max) - SPI_BIT_MASK(min - 1))
    u32 min_speed_hz;
    u32 max_speed_hz;
    u16 flags;
#define SPI_MASTER_HALF_DUPLEX BIT(0)
#define SPI_MASTER_NO_RX BIT(1)
#define SPI_MASTER_NO_TX BIT(2)
#define SPI_MASTER_MUST_RX BIT(3)
#define SPI_MASTER_MUST_TX BIT(4)
    spinlock_t bus_lock_spinlock;
    struct mutex bus_lock_mutex;
    bool bus_lock_flag;
    int (* setup) (struct spi_device *spi);
    int (* transfer) (struct spi_device *spi, struct spi_message *mesg);
    void (* cleanup) (struct spi_device *spi);
    bool (* can_dma) (struct spi_master *master, struct spi_device *spi, struct spi_tr
    bool queued;
    struct kthread_worker kworker;
    struct task_struct * kworker_task;
    struct kthread_work pump_messages;
    spinlock_t queue_lock;
    struct list_head queue;
    struct spi_message * cur_msg;
    bool idling;
    bool busy;
    bool running;
    bool rt;
    bool auto_runtime_pm;
    bool cur_msg_prepared;
    bool cur_msg_mapped;
    struct completion xfer_completion;
    size_t max_dma_len;
    int (* prepare_transfer_hardware) (struct spi_master *master);
    int (* transfer_one_message) (struct spi_master *master, struct spi_message *mesg);
    int (* unprepare_transfer_hardware) (struct spi_master *master);
    int (* prepare_message) (struct spi_master *master, struct spi_message *message);
    int (* unprepare_message) (struct spi_master *master, struct spi_message *message);
```

```
void (* set_cs) (struct spi_device *spi, bool enable);
int (* transfer_one) (struct spi_master *master, struct spi_device *spi, struct spi_message *message);
void (* handle_err) (struct spi_master *master, struct spi_message *message);
int * cs_gpios;
struct spi_statistics statistics;
struct dma_chan * dma_tx;
struct dma_chan * dma_rx;
void * dummy_rx;
void * dummy_tx;
};
```

## Members

dev	device interface to this driver
list	link with the global spi_master list
bus_num	board-specific (and often SOC-specific) identifier for a given SPI controller.
num_chipselect	chipselects are used to distinguish individual SPI slaves, and are numbered from zero to num_chipselects. each slave has a chipselect signal, but it's common that not every chipselect is connected to a slave.
dma_alignment	SPI controller constraint on DMA buffers alignment.
mode_bits	flags understood by this controller driver
bits_per_word_mask	A mask indicating which values of bits_per_word are supported by the driver. Bit n indicates that a bits_per_word n+1 is supported. If set, the SPI core will reject any transfer with an unsupported bits_per_word. If not set, this value is simply ignored, and it's up to the individual driver to perform any validation.
min_speed_hz	Lowest supported transfer speed
max_speed_hz	Highest supported transfer speed
flags	other constraints relevant to this driver
bus_lock_spinlock	spinlock for SPI bus locking
bus_lock_mutex	mutex for SPI bus locking
bus_lock_flag	indicates that the SPI bus is locked for exclusive use
setup	updates the device mode and clocking records used by a device's SPI controller; protocol code may call this. This must fail if an unrecognized or unsupported mode is requested. It's always safe to call this unless transfers are pending on the device whose settings are being modified.
transfer	adds a message to the controller's transfer queue.
cleanup	frees controller-specific state

<code>can_dma</code>	determine whether this master supports DMA
<code>queued</code>	whether this master is providing an internal message queue
<code>kworker</code>	thread struct for message pump
<code>kworker_task</code>	pointer to task for message pump kworker thread
<code>pump_messages</code>	work struct for scheduling work to the message pump
<code>queue_lock</code>	spinlock to synchronise access to message queue
<code>queue</code>	message queue
<code>cur_msg</code>	the currently in-flight message
<code>idling</code>	the device is entering idle state
<code>busy</code>	message pump is busy
<code>running</code>	message pump is running
<code>rt</code>	whether this queue is set to run as a realtime task
<code>auto_runtime_pm</code>	the core should ensure a runtime PM reference is held while the hardware is prepared, using the parent device for the spidev
<code>cur_msg_prepared</code>	<code>spi_prepare_message</code> was called for the currently in-flight message
<code>cur_msg_mapped</code>	message has been mapped for DMA
<code>xfer_completion</code>	used by core <code>transfer_one_message</code>
<code>max_dma_len</code>	Maximum length of a DMA transfer for the device.
<code>prepare_transfer_hardware</code>	a message will soon arrive from the queue so the subsystem requests the driver to prepare the transfer hardware by issuing this call
<code>transfer_one_message</code>	the subsystem calls the driver to transfer a single message while queuing transfers that arrive in the meantime. When the driver is finished with this message, it must call <code>spi_finalize_current_message</code> so the subsystem can issue the next message
<code>unprepare_transfer_hardware</code>	there are currently no more messages on the queue so the subsystem notifies the driver that it may relax the hardware by issuing this call
<code>prepare_message</code>	set up the controller to transfer a single message, for example doing DMA mapping. Called from threaded context.
<code>unprepare_message</code>	undo any work done by <code>prepare_message</code> .
<code>set_cs</code>	set the logic level of the chip select line. May be called from interrupt context.
<code>transfer_one</code>	transfer a single <code>spi_transfer</code> . - return 0 if the transfer is finished, - return 1 if the transfer is still in progress. When the driver is finished with this transfer

	it must call <code>spi_finalize_current_transfer</code> so the subsystem can issue the next transfer. Note: <code>transfer_one</code> and <code>transfer_one_message</code> are mutually exclusive; when both are set, the generic subsystem does not call your <code>transfer_one</code> callback.
<code>handle_err</code>	the subsystem calls the driver to handle an error that occurs in the generic implementation of <code>transfer_one_message</code> .
<code>cs_gpios</code>	Array of GPIOs to use as chip select lines; one per CS number. Any individual value may be <code>-ENOENT</code> for CS lines that are not GPIOs (driven by the SPI controller itself).
<code>statistics</code>	statistics for the <code>spi_master</code>
<code>dma_tx</code>	DMA transmit channel
<code>dma_rx</code>	DMA receive channel
<code>dummy_rx</code>	dummy receive buffer for full-duplex devices
<code>dummy_tx</code>	dummy transmit buffer for full-duplex devices

## Description

Each SPI master controller can communicate with one or more *spi\_device* children. These make a small bus, sharing MOSI, MISO and SCK signals but not chip select signals. Each device may be configured to use a different clock rate, since those shared signals are ignored unless the chip is selected.

The driver for an SPI controller manages access to those devices through a queue of `spi_message` transactions, copying data between CPU memory and an SPI slave device. For each such message it queues, it calls the message's completion function when the transaction completes.

## Name

struct spi\_transfer — a read/write buffer pair

## Synopsis

```
struct spi_transfer {
    const void * tx_buf;
    void * rx_buf;
    unsigned len;
    dma_addr_t tx_dma;
    dma_addr_t rx_dma;
    struct sg_table tx_sg;
    struct sg_table rx_sg;
    unsigned cs_change:1;
    unsigned tx_nbits:3;
    unsigned rx_nbits:3;
#define SPI_NBITS_SINGLE 0x01
#define SPI_NBITS_DUAL 0x02
#define SPI_NBITS_QUAD 0x04
    u8 bits_per_word;
    u16 delay_usecs;
    u32 speed_hz;
    struct list_head transfer_list;
};
```

## Members

tx_buf	data to be written (dma-safe memory), or NULL
rx_buf	data to be read (dma-safe memory), or NULL
len	size of rx and tx buffers (in bytes)
tx_dma	DMA address of tx_buf, if <i>spi_message.is_dma_mapped</i>
rx_dma	DMA address of rx_buf, if <i>spi_message.is_dma_mapped</i>
tx_sg	Scatterlist for transmit, currently not for client use
rx_sg	Scatterlist for receive, currently not for client use
cs_change	affects chipselect after this transfer completes
tx_nbits	number of bits used for writing. If 0 the default (SPI_NBITS_SINGLE) is used.
rx_nbits	number of bits used for reading. If 0 the default (SPI_NBITS_SINGLE) is used.
bits_per_word	select a bits_per_word other than the device default for this transfer. If 0 the default (from <i>spi_device</i> ) is used.
delay_usecs	microseconds to delay after this transfer before (optionally) changing the chipselect status, then starting the next transfer or completing this <i>spi_message</i> .

<code>speed_hz</code>	Select a speed other than the device default for this transfer. If 0 the default (from <i>spi_device</i> ) is used.
<code>transfer_list</code>	transfers are sequenced through <i>spi_message.transfers</i>

## Description

SPI transfers always write the same number of bytes as they read. Protocol drivers should always provide *rx\_buf* and/or *tx\_buf*. In some cases, they may also want to provide DMA addresses for the data being transferred; that may reduce overhead, when the underlying driver uses dma.

If the transmit buffer is null, zeroes will be shifted out while filling *rx\_buf*. If the receive buffer is null, the data shifted in will be discarded. Only “len” bytes shift out (or in). It's an error to try to shift out a partial word. (For example, by shifting out three bytes with word size of sixteen or twenty bits; the former uses two bytes per word, the latter uses four bytes.)

In-memory data values are always in native CPU byte order, translated from the wire byte order (big-endian except with `SPI_LSB_FIRST`). So for example when `bits_per_word` is sixteen, buffers are 2N bytes long (*len* = 2N) and hold N sixteen bit words in CPU byte order.

When the word size of the SPI transfer is not a power-of-two multiple of eight bits, those in-memory words include extra bits. In-memory words are always seen by protocol drivers as right-justified, so the undefined (rx) or unused (tx) bits are always the most significant bits.

All SPI transfers start with the relevant chipselect active. Normally it stays selected until after the last transfer in a message. Drivers can affect the chipselect signal using `cs_change`.

(i) If the transfer isn't the last one in the message, this flag is used to make the chipselect briefly go inactive in the middle of the message. Toggling chipselect in this way may be needed to terminate a chip command, letting a single *spi\_message* perform all of group of chip transactions together.

(ii) When the transfer is the last one in the message, the chip may stay selected until the next transfer. On multi-device SPI busses with nothing blocking messages going to other devices, this is just a performance hint; starting a message to another device deselects this one. But in other cases, this can be used to ensure correctness. Some devices need protocol transactions to be built from a series of *spi\_message* submissions, where the content of one message is determined by the results of previous messages and where the whole transaction ends when the chipselect goes inactive.

When SPI can transfer in 1x, 2x or 4x. It can get this transfer information from device through *tx\_nbits* and *rx\_nbits*. In Bi-direction, these two should both be set. User can set transfer mode with `SPI_NBITS_SINGLE(1x)` `SPI_NBITS_DUAL(2x)` and `SPI_NBITS_QUAD(4x)` to support these three transfer.

The code that submits an *spi\_message* (and its *spi\_transfers*) to the lower layers is responsible for managing its memory. Zero-initialize every field you don't set up explicitly, to insulate against future API updates. After you submit a message and its transfers, ignore them until its completion callback.



## Name

struct spi\_message — one multi-segment SPI transaction

## Synopsis

```
struct spi_message {
    struct list_head transfers;
    struct spi_device * spi;
    unsigned is_dma_mapped:1;
    void (* complete) (void *context);
    void * context;
    unsigned frame_length;
    unsigned actual_length;
    int status;
    struct list_head queue;
    void * state;
};
```

## Members

transfers	list of transfer segments in this transaction
spi	SPI device to which the transaction is queued
is_dma_mapped	if true, the caller provided both dma and cpu virtual addresses for each transfer buffer
complete	called to report transaction completions
context	the argument to <code>complete</code> when it's called
frame_length	the total number of bytes in the message
actual_length	the total number of bytes that were transferred in all successful segments
status	zero for success, else negative errno
queue	for use by whichever driver currently owns the message
state	for use by whichever driver currently owns the message

## Description

A *spi\_message* is used to execute an atomic sequence of data transfers, each represented by a struct *spi\_transfer*. The sequence is “atomic” in the sense that no other *spi\_message* may use that SPI bus until that sequence completes. On some systems, many such sequences can execute as as single programmed DMA transfer. On all systems, these messages are queued, and might complete after transactions to other devices. Messages sent to a given *spi\_device* are always executed in FIFO order.

The code that submits an *spi\_message* (and its *spi\_transfers*) to the lower layers is responsible for managing its memory. Zero-initialize every field you don't set up explicitly, to insulate against future API updates. After you submit a message and its transfers, ignore them until its completion callback.

## Name

`spi_message_init_with_transfers` — Initialize `spi_message` and append transfers

## Synopsis

```
void spi_message_init_with_transfers (struct spi_message * m, struct  
spi_transfer * xfers, unsigned int num_xfers);
```

## Arguments

<i>m</i>	<code>spi_message</code> to be initialized
<i>xfers</i>	An array of spi transfers
<i>num_xfers</i>	Number of items in the xfer array

## Description

This function initializes the given `spi_message` and adds each `spi_transfer` in the given array to the message.

## Name

`spi_write` — SPI synchronous write

## Synopsis

```
int spi_write (struct spi_device * spi, const void * buf, size_t len);
```

## Arguments

*spi* device to which data will be written

*buf* data buffer

*len* data buffer size

## Context

can sleep

## Description

This function writes the buffer *buf*. Callable only from contexts that can sleep.

## Return

zero on success, else a negative error code.

## Name

`spi_read` — SPI synchronous read

## Synopsis

```
int spi_read (struct spi_device * spi, void * buf, size_t len);
```

## Arguments

*spi* device from which data will be read

*buf* data buffer

*len* data buffer size

## Context

can sleep

## Description

This function reads the buffer *buf*. Callable only from contexts that can sleep.

## Return

zero on success, else a negative error code.

## Name

`spi_sync_transfer` — synchronous SPI data transfer

## Synopsis

```
int spi_sync_transfer (struct spi_device * spi, struct spi_transfer *  
xfers, unsigned int num_xfers);
```

## Arguments

*spi*            device with which data will be exchanged

*xfers*        An array of `spi_transfers`

*num\_xfers*   Number of items in the xfer array

## Context

can sleep

## Description

Does a synchronous SPI data transfer of the given `spi_transfer` array.

For more specific semantics see `spi_sync`.

## Return

Return: zero on success, else a negative error code.

## Name

`spi_w8r8` — SPI synchronous 8 bit write followed by 8 bit read

## Synopsis

```
ssize_t spi_w8r8 (struct spi_device * spi, u8 cmd);
```

## Arguments

*spi* device with which data will be exchanged

*cmd* command to be written before data is read back

## Context

can sleep

## Description

Callable only from contexts that can sleep.

## Return

the (unsigned) eight bit number returned by the device, or else a negative error code.

## Name

`spi_w8r16` — SPI synchronous 8 bit write followed by 16 bit read

## Synopsis

```
ssize_t spi_w8r16 (struct spi_device * spi, u8 cmd);
```

## Arguments

*spi* device with which data will be exchanged

*cmd* command to be written before data is read back

## Context

can sleep

## Description

The number is returned in wire-order, which is at least sometimes big-endian.

Callable only from contexts that can sleep.

## Return

the (unsigned) sixteen bit number returned by the device, or else a negative error code.

## Name

`spi_w8r16be` — SPI synchronous 8 bit write followed by 16 bit big-endian read

## Synopsis

```
ssize_t spi_w8r16be (struct spi_device * spi, u8 cmd);
```

## Arguments

*spi* device with which data will be exchanged

*cmd* command to be written before data is read back

## Context

can sleep

## Description

This function is similar to `spi_w8r16`, with the exception that it will convert the read 16 bit data word from big-endian to native endianness.

Callable only from contexts that can sleep.

## Return

the (unsigned) sixteen bit number returned by the device in cpu endianness, or else a negative error code.



## Name

struct spi\_board\_info — board-specific template for a SPI device

## Synopsis

```
struct spi_board_info {
    char modalias[SPI_NAME_SIZE];
    const void * platform_data;
    void * controller_data;
    int irq;
    u32 max_speed_hz;
    u16 bus_num;
    u16 chip_select;
    u16 mode;
};
```

## Members

modalias[SPI_NAME_SIZE]	Initializes spi_device.modalias; identifies the driver.
platform_data	Initializes spi_device.platform_data; the particular data stored there is driver-specific.
controller_data	Initializes spi_device.controller_data; some controllers need hints about hardware setup, e.g. for DMA.
irq	Initializes spi_device.irq; depends on how the board is wired.
max_speed_hz	Initializes spi_device.max_speed_hz; based on limits from the chip datasheet and board-specific signal quality issues.
bus_num	Identifies which spi_master parents the spi_device; unused by spi_new_device, and otherwise depends on board wiring.
chip_select	Initializes spi_device.chip_select; depends on how the board is wired.
mode	Initializes spi_device.mode; based on the chip datasheet, board wiring (some devices support both 3WIRE and standard modes), and possibly presence of an inverter in the chipselect path.

## Description

When adding new SPI devices to the device tree, these structures serve as a partial device template. They hold information which can't always be determined by drivers. Information that probe can establish (such as the default transfer wordsize) is not included here.

These structures are used in two places. Their primary role is to be stored in tables of board-specific device descriptors, which are declared early in board initialization and then used (much later) to populate a controller's device tree after the that controller's driver initializes. A secondary (and atypical) role is as a parameter to spi\_new\_device call, which happens after those controller drivers are active in some dynamic board configuration models.

## Name

`spi_register_board_info` — register SPI devices for a given board

## Synopsis

```
int  spi_register_board_info (struct spi_board_info const * info,
unsigned n);
```

## Arguments

*info*    array of chip descriptors

*n*        how many descriptors are provided

## Context

can sleep

## Description

Board-specific early init code calls this (probably during `arch_initcall`) with segments of the SPI device table. Any device nodes are created later, after the relevant parent SPI controller (`bus_num`) is defined. We keep this table of devices forever, so that reloading a controller driver will not make Linux forget about these hard-wired devices.

Other code can also call this, e.g. a particular add-on board might provide SPI devices through its expansion connector, so code initializing that board would naturally declare its SPI devices.

The board info passed can safely be `__initdata` ... but be careful of any embedded pointers (`platform_data`, etc), they're copied as-is.

## Return

zero on success, else a negative error code.

## Name

`__spi_register_driver` — register a SPI driver

## Synopsis

```
int __spi_register_driver (struct module * owner, struct spi_driver *  
sdrv);
```

## Arguments

*owner*    owner module of the driver to register

*sdrv*     the driver to register

## Context

can sleep

## Return

zero on success, else a negative error code.

## Name

`spi_alloc_device` — Allocate a new SPI device

## Synopsis

```
struct spi_device * spi_alloc_device (struct spi_master * master);
```

## Arguments

*master*    Controller to which device is connected

## Context

can sleep

## Description

Allows a driver to allocate and initialize a `spi_device` without registering it immediately. This allows a driver to directly fill the `spi_device` with device parameters before calling `spi_add_device` on it.

Caller is responsible to call `spi_add_device` on the returned `spi_device` structure to add it to the SPI master. If the caller needs to discard the `spi_device` without adding it, then it should call `spi_dev_put` on it.

## Return

a pointer to the new device, or NULL.

## Name

`spi_add_device` — Add `spi_device` allocated with `spi_alloc_device`

## Synopsis

```
int spi_add_device (struct spi_device * spi);
```

## Arguments

*spi*    `spi_device` to register

## Description

Companion function to `spi_alloc_device`. Devices allocated with `spi_alloc_device` can be added onto the spi bus with this function.

## Return

0 on success; negative `errno` on failure

## Name

`spi_new_device` — instantiate one new SPI device

## Synopsis

```
struct spi_device * spi_new_device (struct spi_master * master, struct  
spi_board_info * chip);
```

## Arguments

*master*    Controller to which device is connected

*chip*     Describes the SPI device

## Context

can sleep

## Description

On typical mainboards, this is purely internal; and it's not needed after board init creates the hard-wired devices. Some development platforms may not be able to use `spi_register_board_info` though, and this is exported so that for example a USB or parport based adapter driver could add devices (which it would learn about out-of-band).

## Return

the new device, or NULL.

## Name

`spi_finalize_current_transfer` — report completion of a transfer

## Synopsis

```
void spi_finalize_current_transfer (struct spi_master * master);
```

## Arguments

*master* the master reporting completion

## Description

Called by SPI drivers using the core `transfer_one_message` implementation to notify it that the current interrupt driven transfer has finished and the next one may be scheduled.

## Name

`spi_get_next_queued_message` — called by driver to check for queued messages

## Synopsis

```
struct spi_message * spi_get_next_queued_message (struct spi_master *  
master);
```

## Arguments

*master* the master to check for queued messages

## Description

If there are more messages in the queue, the next message is returned from this call.

## Return

the next message in the queue, else NULL if the queue is empty.



## Name

`spi_finalize_current_message` — the current message is complete

## Synopsis

```
void spi_finalize_current_message (struct spi_master * master);
```

## Arguments

*master* the master to return the message to

## Description

Called by the driver to notify the core that the message in the front of the queue is complete and can be removed from the queue.

## Name

`spi_alloc_master` — allocate SPI master controller

## Synopsis

```
struct spi_master * spi_alloc_master (struct device * dev, unsigned
size);
```

## Arguments

*dev* the controller, possibly using the `platform_bus`

*size* how much zeroed driver-private data to allocate; the pointer to this memory is in the `driver_data` field of the returned device, accessible with `spi_master_get_devdata`.

## Context

can sleep

## Description

This call is used only by SPI master controller drivers, which are the only ones directly touching chip registers. It's how they allocate an `spi_master` structure, prior to calling `spi_register_master`.

This must be called from context that can sleep.

The caller is responsible for assigning the bus number and initializing the master's methods before calling `spi_register_master`; and (after errors adding the device) calling `spi_master_put` to prevent a memory leak.

## Return

the SPI master structure on success, else `NULL`.

## Name

`spi_register_master` — register SPI master controller

## Synopsis

```
int spi_register_master (struct spi_master * master);
```

## Arguments

*master* initialized master, originally from `spi_alloc_master`

## Context

can sleep

## Description

SPI master controllers connect to their drivers using some non-SPI bus, such as the platform bus. The final stage of `probe` in that code includes calling `spi_register_master` to hook up to this SPI bus glue.

SPI controllers use board specific (often SOC specific) bus numbers, and board-specific addressing for SPI devices combines those numbers with chip select numbers. Since SPI does not directly support dynamic device identification, boards need configuration tables telling which chip is at which address.

This must be called from context that can sleep. It returns zero on success, else a negative error code (dropping the master's refcount). After a successful return, the caller is responsible for calling `spi_unregister_master`.

## Return

zero on success, else a negative error code.

## Name

`devm_spi_register_master` — register managed SPI master controller

## Synopsis

```
int devm_spi_register_master (struct device * dev, struct spi_master
* master);
```

## Arguments

*dev*        device managing SPI master

*master*    initialized master, originally from `spi_alloc_master`

## Context

can sleep

## Description

Register a SPI device as with `spi_register_master` which will automatically be unregister

## Return

zero on success, else a negative error code.

## Name

`spi_unregister_master` — unregister SPI master controller

## Synopsis

```
void spi_unregister_master (struct spi_master * master);
```

## Arguments

*master* the master being unregistered

## Context

can sleep

## Description

This call is used only by SPI master controller drivers, which are the only ones directly touching chip registers.

This must be called from context that can sleep.

## Name

`spi_busnum_to_master` — look up master associated with `bus_num`

## Synopsis

```
struct spi_master * spi_busnum_to_master (u16 bus_num);
```

## Arguments

*bus\_num* the master's bus number

## Context

can sleep

## Description

This call may be used with devices that are registered after arch init time. It returns a refcounted pointer to the relevant `spi_master` (which the caller must release), or NULL if there is no such master registered.

## Return

the SPI master structure on success, else NULL.

## Name

`spi_setup` — setup SPI mode and clock rate

## Synopsis

```
int spi_setup (struct spi_device * spi);
```

## Arguments

*spi* the device whose settings are being modified

## Context

can sleep, and no requests are queued to the device

## Description

SPI protocol drivers may need to update the transfer mode if the device doesn't work with its default. They may likewise need to update clock rates or word sizes from initial values. This function changes those settings, and must be called from a context that can sleep. Except for `SPI_CS_HIGH`, which takes effect immediately, the changes take effect the next time the device is selected and data is transferred to or from it. When this function returns, the spi device is deselected.

Note that this call will fail if the protocol driver specifies an option that the underlying controller or its driver does not support. For example, not all hardware supports wire transfers using nine bit words, LSB-first wire encoding, or active-high chipselects.

## Return

zero on success, else a negative error code.

## Name

`spi_async` — asynchronous SPI transfer

## Synopsis

```
int spi_async (struct spi_device * spi, struct spi_message * message);
```

## Arguments

*spi*            device with which data will be exchanged

*message*       describes the data transfers, including completion callback

## Context

any (irqs may be blocked, etc)

## Description

This call may be used in\_irq and other contexts which can't sleep, as well as from task contexts which can sleep.

The completion callback is invoked in a context which can't sleep. Before that invocation, the value of `message->status` is undefined. When the callback is issued, `message->status` holds either zero (to indicate complete success) or a negative error code. After that callback returns, the driver which issued the transfer request may deallocate the associated memory; it's no longer in use by any SPI core or controller driver code.

Note that although all messages to a `spi_device` are handled in FIFO order, messages may go to different devices in other orders. Some device might be higher priority, or have various “hard” access time requirements, for example.

On detection of any fault during the transfer, processing of the entire message is aborted, and the device is deselected. Until returning from the associated message completion callback, no other `spi_message` queued to that device will be processed. (This rule applies equally to all the synchronous transfer calls, which are wrappers around this core asynchronous primitive.)

## Return

zero on success, else a negative error code.



## Name

`spi_async_locked` — version of `spi_async` with exclusive bus usage

## Synopsis

```
int spi_async_locked (struct spi_device * spi, struct spi_message *  
message);
```

## Arguments

*spi*            device with which data will be exchanged

*message*       describes the data transfers, including completion callback

## Context

any (irqs may be blocked, etc)

## Description

This call may be used in\_irq and other contexts which can't sleep, as well as from task contexts which can sleep.

The completion callback is invoked in a context which can't sleep. Before that invocation, the value of `message->status` is undefined. When the callback is issued, `message->status` holds either zero (to indicate complete success) or a negative error code. After that callback returns, the driver which issued the transfer request may deallocate the associated memory; it's no longer in use by any SPI core or controller driver code.

Note that although all messages to a `spi_device` are handled in FIFO order, messages may go to different devices in other orders. Some device might be higher priority, or have various “hard” access time requirements, for example.

On detection of any fault during the transfer, processing of the entire message is aborted, and the device is deselected. Until returning from the associated message completion callback, no other `spi_message` queued to that device will be processed. (This rule applies equally to all the synchronous transfer calls, which are wrappers around this core asynchronous primitive.)

## Return

zero on success, else a negative error code.

## Name

`spi_sync` — blocking/synchronous SPI data transfers

## Synopsis

```
int spi_sync (struct spi_device * spi, struct spi_message * message);
```

## Arguments

*spi*            device with which data will be exchanged

*message*       describes the data transfers

## Context

can sleep

## Description

This call may only be used from a context that may sleep. The sleep is non-interruptible, and has no timeout. Low-overhead controller drivers may DMA directly into and out of the message buffers.

Note that the SPI device's chip select is active during the message, and then is normally disabled between messages. Drivers for some frequently-used devices may want to minimize costs of selecting a chip, by leaving it selected in anticipation that the next message will go to the same chip. (That may increase power usage.)

Also, the caller is guaranteeing that the memory associated with the message will not be freed before this call returns.

## Return

zero on success, else a negative error code.

## Name

`spi_sync_locked` — version of `spi_sync` with exclusive bus usage

## Synopsis

```
int spi_sync_locked (struct spi_device * spi, struct spi_message *  
message);
```

## Arguments

*spi*            device with which data will be exchanged

*message*       describes the data transfers

## Context

can sleep

## Description

This call may only be used from a context that may sleep. The sleep is non-interruptible, and has no timeout. Low-overhead controller drivers may DMA directly into and out of the message buffers.

This call should be used by drivers that require exclusive access to the SPI bus. It has to be preceded by a `spi_bus_lock` call. The SPI bus must be released by a `spi_bus_unlock` call when the exclusive access is over.

## Return

zero on success, else a negative error code.

## Name

`spi_bus_lock` — obtain a lock for exclusive SPI bus usage

## Synopsis

```
int spi_bus_lock (struct spi_master * master);
```

## Arguments

*master* SPI bus master that should be locked for exclusive bus access

## Context

can sleep

## Description

This call may only be used from a context that may sleep. The sleep is non-interruptible, and has no timeout.

This call should be used by drivers that require exclusive access to the SPI bus. The SPI bus must be released by a `spi_bus_unlock` call when the exclusive access is over. Data transfer must be done by `spi_sync_locked` and `spi_async_locked` calls when the SPI bus lock is held.

## Return

always zero.

## Name

`spi_bus_unlock` — release the lock for exclusive SPI bus usage

## Synopsis

```
int spi_bus_unlock (struct spi_master * master);
```

## Arguments

*master* SPI bus master that was locked for exclusive bus access

## Context

can sleep

## Description

This call may only be used from a context that may sleep. The sleep is non-interruptible, and has no timeout.

This call releases an SPI bus lock previously obtained by an `spi_bus_lock` call.

## Return

always zero.

## Name

`spi_write_then_read` — SPI synchronous write followed by read

## Synopsis

```
int spi_write_then_read (struct spi_device * spi, const void * txbuf,  
unsigned n_tx, void * rxbuf, unsigned n_rx);
```

## Arguments

*spi*      device with which data will be exchanged

*txbuf*    data to be written (need not be dma-safe)

*n\_tx*     size of txbuf, in bytes

*rxbuf*    buffer into which data will be read (need not be dma-safe)

*n\_rx*     size of rxbuf, in bytes

## Context

can sleep

## Description

This performs a half duplex MicroWire style transaction with the device, sending txbuf and then reading rxbuf. The return value is zero for success, else a negative errno status code. This call may only be used from a context that may sleep.

Parameters to this routine are always copied using a small buffer; portable code should never use this for more than 32 bytes. Performance-sensitive or bulk transfer code should instead use `spi_{async, sync}()` calls with dma-safe buffers.

## Return

zero on success, else a negative error code.

---

# Chapter 11. I<sup>2</sup>C and SMBus Subsystem

I<sup>2</sup>C (or without fancy typography, "I2C") is an acronym for the "Inter-IC" bus, a simple bus protocol which is widely used where low data rate communications suffice. Since it's also a licensed trademark, some vendors use another name (such as "Two-Wire Interface", TWI) for the same bus. I2C only needs two signals (SCL for clock, SDA for data), conserving board real estate and minimizing signal quality issues. Most I2C devices use seven bit addresses, and bus speeds of up to 400 kHz; there's a high speed extension (3.4 MHz) that's not yet found wide use. I2C is a multi-master bus; open drain signaling is used to arbitrate between masters, as well as to handshake and to synchronize clocks from slower clients.

The Linux I2C programming interfaces support only the master side of bus interactions, not the slave side. The programming interface is structured around two kinds of driver, and two kinds of device. An I2C "Adapter Driver" abstracts the controller hardware; it binds to a physical device (perhaps a PCI device or platform\_device) and exposes a struct `i2c_adapter` representing each I2C bus segment it manages. On each I2C bus segment will be I2C devices represented by a struct `i2c_client`. Those devices will be bound to a struct `i2c_driver`, which should follow the standard Linux driver model. (At this writing, a legacy model is more widely used.) There are functions to perform various I2C protocol operations; at this writing all such functions are usable only from task context.

The System Management Bus (SMBus) is a sibling protocol. Most SMBus systems are also I2C conformant. The electrical constraints are tighter for SMBus, and it standardizes particular protocol messages and idioms. Controllers that support I2C can also support most SMBus operations, but SMBus controllers don't support all the protocol options that an I2C controller will. There are functions to perform various SMBus protocol operations, either using I2C primitives or by issuing SMBus commands to `i2c_adapter` devices which don't support those I2C operations.

## Name

struct i2c\_driver — represent an I2C device driver

## Synopsis

```
struct i2c_driver {
    unsigned int class;
    int (* attach_adapter) (struct i2c_adapter *);
    int (* probe) (struct i2c_client *, const struct i2c_device_id *);
    int (* remove) (struct i2c_client *);
    void (* shutdown) (struct i2c_client *);
    void (* alert) (struct i2c_client *, unsigned int data);
    int (* command) (struct i2c_client *client, unsigned int cmd, void *arg);
    struct device_driver driver;
    const struct i2c_device_id * id_table;
    int (* detect) (struct i2c_client *, struct i2c_board_info *);
    const unsigned short * address_list;
    struct list_head clients;
};
```

## Members

class	What kind of i2c device we instantiate (for detect)
attach_adapter	Callback for bus addition (deprecated)
probe	Callback for device binding
remove	Callback for device unbinding
shutdown	Callback for device shutdown
alert	Alert callback, for example for the SMBus alert protocol
command	Callback for bus-wide signaling (optional)
driver	Device driver model driver
id_table	List of I2C devices supported by this driver
detect	Callback for device detection
address_list	The I2C addresses to probe (for detect)
clients	List of detected clients we created (for i2c-core use only)

## Description

The driver.owner field should be set to the module owner of this driver. The driver.name field should be set to the name of this driver.

For automatic device detection, both *detect* and *address\_list* must be defined. *class* should also be set, otherwise only devices forced with module parameters will be created. The detect function must fill



at least the name field of the `i2c_board_info` structure it is handed upon successful detection, and possibly also the flags field.

If *detect* is missing, the driver will still work fine for enumerated devices. Detected devices simply won't be supported. This is expected for the many I2C/SMBus devices which can't be detected reliably, and the ones which can always be enumerated in practice.

The `i2c_client` structure which is handed to the *detect* callback is not a real `i2c_client`. It is initialized just enough so that you can call `i2c_smbus_read_byte_data` and friends on it. Don't do anything else with it. In particular, calling `dev_dbg` and friends on it is not allowed.

## Name

struct i2c\_client — represent an I2C slave device

## Synopsis

```
struct i2c_client {
    unsigned short flags;
    unsigned short addr;
    char name[I2C_NAME_SIZE];
    struct i2c_adapter * adapter;
    struct device dev;
    int irq;
    struct list_head detected;
#ifdef IS_ENABLED(CONFIG_I2C_SLAVE)
    i2c_slave_cb_t slave_cb;
#endif
};
```

## Members

flags	I2C_CLIENT_TEN indicates the device uses a ten bit chip address; I2C_CLIENT_PEC indicates it uses SMBus Packet Error Checking
addr	Address used on the I2C bus connected to the parent adapter.
name[I2C_NAME_SIZE]	Indicates the type of the device, usually a chip name that's generic enough to hide second-sourcing and compatible revisions.
adapter	manages the bus segment hosting this I2C device
dev	Driver model device node for the slave.
irq	indicates the IRQ generated by this device (if any)
detected	member of an i2c_driver.clients list or i2c-core's userspace_devices list
slave_cb	Callback when I2C slave mode of an adapter is used. The adapter calls it to pass on slave events to the slave driver.

## Description

An i2c\_client identifies a single device (i.e. chip) connected to an i2c bus. The behaviour exposed to Linux is defined by the driver managing the device.

## Name

struct i2c\_board\_info — template for device creation

## Synopsis

```
struct i2c_board_info {
    char type[I2C_NAME_SIZE];
    unsigned short flags;
    unsigned short addr;
    void * platform_data;
    struct dev_archdata * archdata;
    struct device_node * of_node;
    struct fwnode_handle * fwnode;
    int irq;
};
```

## Members

type[I2C_NAME_SIZE]	chip type, to initialize i2c_client.name
flags	to initialize i2c_client.flags
addr	stored in i2c_client.addr
platform_data	stored in i2c_client.dev.platform_data
archdata	copied into i2c_client.dev.archdata
of_node	pointer to OpenFirmware device node
fwnode	device node supplied by the platform firmware
irq	stored in i2c_client.irq

## Description

I2C doesn't actually support hardware probing, although controllers and devices may be able to use I2C\_SMBUS\_QUICK to tell whether or not there's a device at a given address. Drivers commonly need more information than that, such as chip type, configuration, associated IRQ, and so on.

i2c\_board\_info is used to build tables of information listing I2C devices that are present. This information is used to grow the driver model tree. For mainboards this is done statically using i2c\_register\_board\_info; bus numbers identify adapters that aren't yet available. For add-on boards, i2c\_new\_device does this dynamically with the adapter already known.

## Name

I2C\_BOARD\_INFO — macro used to list an i2c device and its address

## Synopsis

```
I2C_BOARD_INFO ( dev_type, dev_addr );
```

## Arguments

*dev\_type* identifies the device type

*dev\_addr* the device's address on the bus.

## Description

This macro initializes essential fields of a struct `i2c_board_info`, declaring what has been provided on a particular board. Optional fields (such as associated `irq`, or device-specific `platform_data`) are provided using conventional syntax.

## Name

struct i2c\_algorithm — represent I2C transfer method

## Synopsis

```
struct i2c_algorithm {
    int (* master_xfer) (struct i2c_adapter *adap, struct i2c_msg *msgs,int num);
    int (* smbus_xfer) (struct i2c_adapter *adap, ul6 addr,unsigned short flags, cha
    u32 (* functionality) (struct i2c_adapter *);
#ifdef IS_ENABLED(CONFIG_I2C_SLAVE)
    int (* reg_slave) (struct i2c_client *client);
    int (* unreg_slave) (struct i2c_client *client);
#endif
};
```

## Members

master_xfer	Issue a set of i2c transactions to the given I2C adapter defined by the msgs array, with num messages available to transfer via the adapter specified by adap.
smbus_xfer	Issue smbus transactions to the given I2C adapter. If this is not present, then the bus layer will try and convert the SMBus calls into I2C transfers instead.
functionality	Return the flags that this algorithm/adapter pair supports from the I2C_FUNC_* flags.
reg_slave	Register given client to I2C slave mode of this adapter
unreg_slave	Unregister given client from I2C slave mode of this adapter

## The following structs are for those who like to implement new bus drivers

i2c\_algorithm is the interface to a class of hardware solutions which can be addressed using the same bus algorithms - i.e. bit-banging or the PCF8584 to name two of the most common.

The return codes from the *master\_xfer* field should indicate the type of error code that occurred during the transfer, as documented in the kernel Documentation file Documentation/i2c/fault-codes.

## Name

struct i2c\_bus\_recovery\_info — I2C bus recovery information

## Synopsis

```
struct i2c_bus_recovery_info {
    int (* recover_bus) (struct i2c_adapter *);
    int (* get_scl) (struct i2c_adapter *);
    void (* set_scl) (struct i2c_adapter *, int val);
    int (* get_sda) (struct i2c_adapter *);
    void (* prepare_recovery) (struct i2c_adapter *);
    void (* unprepare_recovery) (struct i2c_adapter *);
    int scl_gpio;
    int sda_gpio;
};
```

## Members

recover_bus	Recover routine. Either pass driver's <code>recover_bus</code> routine, or <code>i2c_generic_scl_recovery</code> or <code>i2c_generic_gpio_recovery</code> .
get_scl	This gets current value of SCL line. Mandatory for generic SCL recovery. Used internally for generic GPIO recovery.
set_scl	This sets/clears SCL line. Mandatory for generic SCL recovery. Used internally for generic GPIO recovery.
get_sda	This gets current value of SDA line. Optional for generic SCL recovery. Used internally, if <code>sda_gpio</code> is a valid GPIO, for generic GPIO recovery.
prepare_recovery	This will be called before starting recovery. Platform may configure padmux here for SDA/SCL line or something else they want.
unprepare_recovery	This will be called after completing recovery. Platform may configure padmux here for SDA/SCL line or something else they want.
scl_gpio	gpio number of the SCL line. Only required for GPIO recovery.
sda_gpio	gpio number of the SDA line. Only required for GPIO recovery.

## Name

struct i2c\_adapter\_quirks — describe flaws of an i2c adapter

## Synopsis

```
struct i2c_adapter_quirks {  
    u64 flags;  
    int max_num_msgs;  
    u16 max_write_len;  
    u16 max_read_len;  
    u16 max_comb_1st_msg_len;  
    u16 max_comb_2nd_msg_len;  
};
```

## Members

flags	see I2C_AQ_* for possible flags and read below
max_num_msgs	maximum number of messages per transfer
max_write_len	maximum length of a write message
max_read_len	maximum length of a read message
max_comb_1st_msg_len	maximum length of the first msg in a combined message
max_comb_2nd_msg_len	maximum length of the second msg in a combined message

## Note about combined messages

Some I2C controllers can only send one message per transfer, plus something called combined message or write-then-read. This is (usually) a small write message followed by a read message and barely enough to access register based devices like EEPROMs. There is a flag to support this mode. It implies `max_num_msg = 2` and does the length checks with `max_comb_*_len` because combined message mode usually has its own limitations. Because of HW implementations, some controllers can actually do write-then-anything or other variants. To support that, write-then-read has been broken out into smaller bits like write-first and read-second which can be combined as needed.

## Name

`module_i2c_driver` — Helper macro for registering a I2C driver

## Synopsis

```
module_i2c_driver ( __i2c_driver);
```

## Arguments

`__i2c_driver` i2c\_driver struct

## Description

Helper macro for I2C drivers which do not do anything special in module init/exit. This eliminates a lot of boilerplate. Each module may only use this macro once, and calling it replaces `module_init` and `module_exit`



## Name

`i2c_register_board_info` — statically declare I2C devices

## Synopsis

```
int i2c_register_board_info (int busnum, struct i2c_board_info const *  
info, unsigned len);
```

## Arguments

*busnum* identifies the bus to which these devices belong

*info* vector of i2c device descriptors

*len* how many descriptors in the vector; may be zero to reserve the specified bus number.

## Description

Systems using the Linux I2C driver stack can declare tables of board info while they initialize. This should be done in board-specific init code near `arch_initcall` time, or equivalent, before any I2C adapter driver is registered. For example, mainboard init code could define several devices, as could the init code for each daughtercard in a board stack.

The I2C devices will be created later, after the adapter for the relevant bus has been registered. After that moment, standard driver model tools are used to bind “new style” I2C drivers to the devices. The bus number for any device declared using this routine is not available for dynamic allocation.

The board info passed can safely be `__initdata`, but be careful of embedded pointers (for `platform_data`, functions, etc) since that won't be copied.

## Name

`i2c_verify_client` — return parameter as `i2c_client`, or `NULL`

## Synopsis

```
struct i2c_client * i2c_verify_client (struct device * dev);
```

## Arguments

*dev* device, probably from some driver model iterator

## Description

When traversing the driver model tree, perhaps using driver model iterators like `device_for_each_child()`, you can't assume very much about the nodes you find. Use this function to avoid oopses caused by wrongly treating some non-I2C device as an `i2c_client`.

## Name

`i2c_lock_adapter` — Get exclusive access to an I2C bus segment

## Synopsis

```
void i2c_lock_adapter (struct i2c_adapter * adapter);
```

## Arguments

*adapter*    Target I2C bus segment

## Name

`i2c_unlock_adapter` — Release exclusive access to an I2C bus segment

## Synopsis

```
void i2c_unlock_adapter (struct i2c_adapter * adapter);
```

## Arguments

*adapter*    Target I2C bus segment

## Name

`i2c_new_device` — instantiate an i2c device

## Synopsis

```
struct i2c_client * i2c_new_device (struct i2c_adapter * adap, struct  
i2c_board_info const * info);
```

## Arguments

*adap* the adapter managing the device

*info* describes one I2C device; `bus_num` is ignored

## Context

can sleep

## Description

Create an i2c device. Binding is handled through driver model `probe/remove` methods. A driver may be bound to this device when we return from this function, or any later moment (e.g. maybe hotplugging will load the driver module). This call is not appropriate for use by mainboard initialization logic, which usually runs during an `arch_initcall` long before any `i2c_adapter` could exist.

This returns the new i2c client, which may be saved for later use with `i2c_unregister_device`; or `NULL` to indicate an error.

## Name

`i2c_unregister_device` — reverse effect of `i2c_new_device`

## Synopsis

```
void i2c_unregister_device (struct i2c_client * client);
```

## Arguments

*client*    value returned from `i2c_new_device`

## Context

can sleep

## Name

`i2c_new_dummy` — return a new i2c device bound to a dummy driver

## Synopsis

```
struct i2c_client * i2c_new_dummy (struct i2c_adapter * adapter, u16  
address);
```

## Arguments

*adapter*    the adapter managing the device

*address*    seven bit address to be used

## Context

can sleep

## Description

This returns an I2C client bound to the “dummy” driver, intended for use with devices that consume multiple addresses. Examples of such chips include various EEPROMS (like 24c04 and 24c08 models).

These dummy devices have two main uses. First, most I2C and SMBus calls except `i2c_transfer` need a client handle; the dummy will be that handle. And second, this prevents the specified address from being bound to a different driver.

This returns the new i2c client, which should be saved for later use with `i2c_unregister_device`; or NULL to indicate an error.

## Name

`i2c_verify_adapter` — return parameter as `i2c_adapter` or `NULL`

## Synopsis

```
struct i2c_adapter * i2c_verify_adapter (struct device * dev);
```

## Arguments

*dev* device, probably from some driver model iterator

## Description

When traversing the driver model tree, perhaps using driver model iterators like `device_for_each_child()`, you can't assume very much about the nodes you find. Use this function to avoid oopses caused by wrongly treating some non-I2C device as an `i2c_adapter`.



## Name

`i2c_add_adapter` — declare i2c adapter, use dynamic bus number

## Synopsis

```
int i2c_add_adapter (struct i2c_adapter * adapter);
```

## Arguments

*adapter*    the adapter to add

## Context

can sleep

## Description

This routine is used to declare an I2C adapter when its bus number doesn't matter or when its bus number is specified by an dt alias. Examples of bases when the bus number doesn't matter: I2C adapters dynamically added by USB links or PCI plugin cards.

When this returns zero, a new bus number was allocated and stored in `adap->nr`, and the specified adapter became available for clients. Otherwise, a negative `errno` value is returned.

## Name

`i2c_add_numbered_adapter` — declare i2c adapter, use static bus number

## Synopsis

```
int i2c_add_numbered_adapter (struct i2c_adapter * adap);
```

## Arguments

*adap* the adapter to register (with `adap->nr` initialized)

## Context

can sleep

## Description

This routine is used to declare an I2C adapter when its bus number matters. For example, use it for I2C adapters from system-on-chip CPUs, or otherwise built in to the system's mainboard, and where `i2c_board_info` is used to properly configure I2C devices.

If the requested bus number is set to -1, then this function will behave identically to `i2c_add_adapter`, and will dynamically assign a bus number.

If no devices have pre-been declared for this bus, then be sure to register the adapter before any dynamically allocated ones. Otherwise the required bus ID may not be available.

When this returns zero, the specified adapter became available for clients using the bus number provided in `adap->nr`. Also, the table of I2C devices pre-declared using `i2c_register_board_info` is scanned, and the appropriate driver model device nodes are created. Otherwise, a negative `errno` value is returned.

## Name

`i2c_del_adapter` — unregister I2C adapter

## Synopsis

```
void i2c_del_adapter (struct i2c_adapter * adap);
```

## Arguments

*adap* the adapter being unregistered

## Context

can sleep

## Description

This unregisters an I2C adapter which was previously registered by *i2c\_add\_adapter* or *i2c\_add\_numbered\_adapter*.

## Name

`i2c_del_driver` — unregister I2C driver

## Synopsis

```
void i2c_del_driver (struct i2c_driver * driver);
```

## Arguments

*driver* the driver being unregistered

## Context

can sleep

## Name

`i2c_use_client` — increments the reference count of the i2c client structure

## Synopsis

```
struct i2c_client * i2c_use_client (struct i2c_client * client);
```

## Arguments

*client* the client being referenced

## Description

Each live reference to a client should be refcounted. The driver model does that automatically as part of driver binding, so that most drivers don't

## need to do this explicitly

they hold a reference until they're unbound from the device.

A pointer to the client with the incremented reference counter is returned.

## Name

`i2c_release_client` — release a use of the i2c client structure

## Synopsis

```
void i2c_release_client (struct i2c_client * client);
```

## Arguments

*client* the client being no longer referenced

## Description

Must be called when a user of a client is finished with it.

## Name

`__i2c_transfer` — unlocked flavor of `i2c_transfer`

## Synopsis

```
int __i2c_transfer (struct i2c_adapter * adap, struct i2c_msg * msgs,
int num);
```

## Arguments

*adap*    Handle to I2C bus

*msgs*    One or more messages to execute before STOP is issued to terminate the operation; each message begins with a START.

*num*    Number of messages to be executed.

## Description

Returns negative errno, else the number of messages executed.

Adapter lock must be held when calling this function. No debug logging takes place. `adap->algo->master_xfer` existence isn't checked.

## Name

`i2c_transfer` — execute a single or combined I2C message

## Synopsis

```
int i2c_transfer (struct i2c_adapter * adap, struct i2c_msg * msgs,
int num);
```

## Arguments

*adap*    Handle to I2C bus

*msgs*    One or more messages to execute before STOP is issued to terminate the operation; each message begins with a START.

*num*    Number of messages to be executed.

## Description

Returns negative errno, else the number of messages executed.

Note that there is no requirement that each message be sent to the same slave address, although that is the most common model.



## Name

`i2c_master_send` — issue a single I2C message in master transmit mode

## Synopsis

```
int i2c_master_send (const struct i2c_client * client, const char *  
buf, int count);
```

## Arguments

*client*    Handle to slave device

*buf*        Data that will be written to the slave

*count*     How many bytes to write, must be less than 64k since `msg.len` is `u16`

## Description

Returns negative `errno`, or else the number of bytes written.

## Name

`i2c_master_recv` — issue a single I2C message in master receive mode

## Synopsis

```
int i2c_master_recv (const struct i2c_client * client, char * buf, int  
count);
```

## Arguments

*client*    Handle to slave device

*buf*        Where to store data read from slave

*count*      How many bytes to read, must be less than 64k since msg.len is u16

## Description

Returns negative errno, or else the number of bytes read.

## Name

i2c\_smbus\_read\_byte — SMBus “receive byte” protocol

## Synopsis

```
s32 i2c_smbus_read_byte (const struct i2c_client * client);
```

## Arguments

*client*    Handle to slave device

## Description

This executes the SMBus “receive byte” protocol, returning negative errno else the byte received from the device.

## Name

i2c\_smbus\_write\_byte — SMBus “send byte” protocol

## Synopsis

```
s32 i2c_smbus_write_byte (const struct i2c_client * client, u8 value);
```

## Arguments

*client*    Handle to slave device

*value*    Byte to be sent

## Description

This executes the SMBus “send byte” protocol, returning negative errno else zero on success.

## Name

`i2c_smbus_read_byte_data` — SMBus “read byte” protocol

## Synopsis

```
s32 i2c_smbus_read_byte_data (const struct i2c_client * client, u8
command);
```

## Arguments

*client*     Handle to slave device

*command*   Byte interpreted by slave

## Description

This executes the SMBus “read byte” protocol, returning negative `errno` else a data byte received from the device.

## Name

`i2c_smbus_write_byte_data` — SMBus “write byte” protocol

## Synopsis

```
s32 i2c_smbus_write_byte_data (const struct i2c_client * client, u8
command, u8 value);
```

## Arguments

*client*     Handle to slave device

*command*   Byte interpreted by slave

*value*      Byte being written

## Description

This executes the SMBus “write byte” protocol, returning negative `errno` else zero on success.

## Name

`i2c_smbus_read_word_data` — SMBus “read word” protocol

## Synopsis

```
s32 i2c_smbus_read_word_data (const struct i2c_client * client, u8
command);
```

## Arguments

*client*     Handle to slave device

*command*   Byte interpreted by slave

## Description

This executes the SMBus “read word” protocol, returning negative `errno` else a 16-bit unsigned “word” received from the device.

## Name

`i2c_smbus_write_word_data` — SMBus “write word” protocol

## Synopsis

```
s32 i2c_smbus_write_word_data (const struct i2c_client * client, u8
command, u16 value);
```

## Arguments

*client*     Handle to slave device

*command*   Byte interpreted by slave

*value*      16-bit “word” being written

## Description

This executes the SMBus “write word” protocol, returning negative `errno` else zero on success.



## Name

`i2c_smbus_read_block_data` — SMBus “block read” protocol

## Synopsis

```
s32 i2c_smbus_read_block_data (const struct i2c_client * client, u8
command, u8 * values);
```

## Arguments

*client*     Handle to slave device

*command*   Byte interpreted by slave

*values*     Byte array into which data will be read; big enough to hold the data returned by the slave.  
SMBus allows at most 32 bytes.

## Description

This executes the SMBus “block read” protocol, returning negative `errno` else the number of data bytes in the slave's response.

Note that using this function requires that the client's adapter support the `I2C_FUNC_SMBUS_READ_BLOCK_DATA` functionality. Not all adapter drivers support this; its emulation through I2C messaging relies on a specific mechanism (`I2C_M_RECV_LEN`) which may not be implemented.

## Name

`i2c_smbus_write_block_data` — SMBus “block write” protocol

## Synopsis

```
s32 i2c_smbus_write_block_data (const struct i2c_client * client, u8
command, u8 length, const u8 * values);
```

## Arguments

*client*     Handle to slave device

*command*   Byte interpreted by slave

*length*     Size of data block; SMBus allows at most 32 bytes

*values*     Byte array which will be written.

## Description

This executes the SMBus “block write” protocol, returning negative `errno` else zero on success.

## Name

`i2c_smbus_xfer` — execute SMBus protocol operations

## Synopsis

```
s32 i2c_smbus_xfer (struct i2c_adapter * adapter, u16 addr, unsigned
short flags, char read_write, u8 command, int protocol, union
i2c_smbus_data * data);
```

## Arguments

<i>adapter</i>	Handle to I2C bus
<i>addr</i>	Address of SMBus slave on that bus
<i>flags</i>	I2C_CLIENT_* flags (usually zero or I2C_CLIENT_PEC)
<i>read_write</i>	I2C_SMBUS_READ or I2C_SMBUS_WRITE
<i>command</i>	Byte interpreted by slave, for protocols which use such bytes
<i>protocol</i>	SMBus protocol operation to execute, such as I2C_SMBUS_PROC_CALL
<i>data</i>	Data to be read or written

## Description

This executes an SMBus protocol operation, and returns a negative errno code else zero on success.

## Name

`i2c_smbus_read_i2c_block_data_or_emulated` — read block or emulate

## Synopsis

```
s32 i2c_smbus_read_i2c_block_data_or_emulated (const struct i2c_client
* client, u8 command, u8 length, u8 * values);
```

## Arguments

<i>client</i>	Handle to slave device
<i>command</i>	Byte interpreted by slave
<i>length</i>	Size of data block; SMBus allows at most I2C_SMBUS_BLOCK_MAX bytes
<i>values</i>	Byte array into which data will be read; big enough to hold the data returned by the slave. SMBus allows at most I2C_SMBUS_BLOCK_MAX bytes.

## Description

This executes the SMBus “block read” protocol if supported by the adapter. If block read is not supported, it emulates it using either word or byte read protocols depending on availability.

The addresses of the I2C slave device that are accessed with this function must be mapped to a linear region, so that a block read will have the same effect as a byte read. Before using this function you must double-check if the I2C slave does support exchanging a block transfer with a byte transfer.

---

# Chapter 12. High Speed Synchronous Serial Interface (HSI)

High Speed Synchronous Serial Interface (HSI) is a serial interface mainly used for connecting application engines (APE) with cellular modem engines (CMT) in cellular handsets. HSI provides multiplexing for up to 16 logical channels, low-latency and full duplex communication.

## Name

struct hsi\_channel — channel resource used by the hsi clients

## Synopsis

```
struct hsi_channel {  
    unsigned int id;  
    const char * name;  
};
```

## Members

id	Channel number
name	Channel name

## Name

struct hsi\_config — Configuration for RX/TX HSI modules

## Synopsis

```
struct hsi_config {  
    unsigned int mode;  
    struct hsi_channel * channels;  
    unsigned int num_channels;  
    unsigned int num_hw_channels;  
    unsigned int speed;  
    union {unnamed_union};  
};
```

## Members

mode	Bit transmission mode (STREAM or FRAME)
channels	Channel resources used by the client
num_channels	Number of channel resources
num_hw_channels	Number of channels the transceiver is configured for [1..16]
speed	Max bit transmission speed (Kbit/s)
{unnamed_union}	anonymous

## Name

struct hsi\_board\_info — HSI client board info

## Synopsis

```
struct hsi_board_info {  
    const char * name;  
    unsigned int hsi_id;  
    unsigned int port;  
    struct hsi_config tx_cfg;  
    struct hsi_config rx_cfg;  
    void * platform_data;  
    struct dev_archdata * archdata;  
};
```

## Members

name	Name for the HSI device
hsi_id	HSI controller id where the client sits
port	Port number in the controller where the client sits
tx_cfg	HSI TX configuration
rx_cfg	HSI RX configuration
platform_data	Platform related data
archdata	Architecture-dependent device data



## Name

struct hsi\_client — HSI client attached to an HSI port

## Synopsis

```
struct hsi_client {  
    struct device device;  
    struct hsi_config tx_cfg;  
    struct hsi_config rx_cfg;  
};
```

## Members

device	Driver model representation of the device
tx_cfg	HSI TX configuration
rx_cfg	HSI RX configuration

## Name

struct hsi\_client\_driver — Driver associated to an HSI client

## Synopsis

```
struct hsi_client_driver {  
    struct device_driver driver;  
};
```

## Members

driver      Driver model representation of the driver

## Name

struct hsi\_msg — HSI message descriptor

## Synopsis

```
struct hsi_msg {
    struct list_head link;
    struct hsi_client * cl;
    struct sg_table sgt;
    void * context;
    void (* complete) (struct hsi_msg *msg);
    void (* destructor) (struct hsi_msg *msg);
    int status;
    unsigned int actual_len;
    unsigned int channel;
    unsigned int ttype:1;
    unsigned int break_frame:1;
};
```

## Members

link	Free to use by the current descriptor owner
cl	HSI device client that issues the transfer
sgt	Head of the scatterlist array
context	Client context data associated to the transfer
complete	Transfer completion callback
destructor	Destructor to free resources when flushing
status	Status of the transfer when completed
actual_len	Actual length of data transferred on completion
channel	Channel were to TX/RX the message
ttype	Transfer type (TX if set, RX otherwise)
break_frame	if true HSI will send/receive a break frame. Data buffers are ignored in the request.

## Name

struct hsi\_port — HSI port device

## Synopsis

```
struct hsi_port {
    struct device device;
    struct hsi_config tx_cfg;
    struct hsi_config rx_cfg;
    unsigned int num;
    unsigned int shared:1;
    int claimed;
    struct mutex lock;
    int (* async) (struct hsi_msg *msg);
    int (* setup) (struct hsi_client *cl);
    int (* flush) (struct hsi_client *cl);
    int (* start_tx) (struct hsi_client *cl);
    int (* stop_tx) (struct hsi_client *cl);
    int (* release) (struct hsi_client *cl);
    struct atomic_notifier_head n_head;
};
```

## Members

device	Driver model representation of the device
tx_cfg	Current TX path configuration
rx_cfg	Current RX path configuration
num	Port number
shared	Set when port can be shared by different clients
claimed	Reference count of clients which claimed the port
lock	Serialize port claim
async	Asynchronous transfer callback
setup	Callback to set the HSI client configuration
flush	Callback to clean the HW state and destroy all pending transfers
start_tx	Callback to inform that a client wants to TX data
stop_tx	Callback to inform that a client no longer wishes to TX data
release	Callback to inform that a client no longer uses the port
n_head	Notifier chain for signaling port events to the clients.

## Name

struct hsi\_controller — HSI controller device

## Synopsis

```
struct hsi_controller {  
    struct device device;  
    struct module * owner;  
    unsigned int id;  
    unsigned int num_ports;  
    struct hsi_port ** port;  
};
```

## Members

device	Driver model representation of the device
owner	Pointer to the module owning the controller
id	HSI controller ID
num_ports	Number of ports in the HSI controller
port	Array of HSI ports

## Name

`hsi_id` — Get HSI controller ID associated to a client

## Synopsis

```
unsigned int hsi_id (struct hsi_client * cl);
```

## Arguments

*cl* Pointer to a HSI client

## Description

Return the controller id where the client is attached to

## Name

`hsi_port_id` — Gets the port number a client is attached to

## Synopsis

```
unsigned int hsi_port_id (struct hsi_client * cl);
```

## Arguments

*cl* Pointer to HSI client

## Description

Return the port number associated to the client

## Name

`hsi_setup` — Configure the client's port

## Synopsis

```
int hsi_setup (struct hsi_client * cl);
```

## Arguments

*cl* Pointer to the HSI client

## Description

When sharing ports, clients should either relay on a single client setup or have the same setup for all of them.

Return `-errno` on failure, 0 on success



## Name

`hsi_flush` — Flush all pending transactions on the client's port

## Synopsis

```
int hsi_flush (struct hsi_client * cl);
```

## Arguments

*cl*    Pointer to the HSI client

## Description

This function will destroy all pending `hsi_msg` in the port and reset the HW port so it is ready to receive and transmit from a clean state.

Return `-errno` on failure, 0 on success

## Name

`hsi_async_read` — Submit a read transfer

## Synopsis

```
int hsi_async_read (struct hsi_client * cl, struct hsi_msg * msg);
```

## Arguments

*cl*     Pointer to the HSI client

*msg*   HSI message descriptor of the transfer

## Description

Return `-errno` on failure, 0 on success

## Name

`hsi_async_write` — Submit a write transfer

## Synopsis

```
int hsi_async_write (struct hsi_client * cl, struct hsi_msg * msg);
```

## Arguments

*cl*     Pointer to the HSI client

*msg*   HSI message descriptor of the transfer

## Description

Return `-errno` on failure, 0 on success

## Name

`hsi_start_tx` — Signal the port that the client wants to start a TX

## Synopsis

```
int hsi_start_tx (struct hsi_client * cl);
```

## Arguments

*cl* Pointer to the HSI client

## Description

Return `-errno` on failure, 0 on success

## Name

`hsi_stop_tx` — Signal the port that the client no longer wants to transmit

## Synopsis

```
int hsi_stop_tx (struct hsi_client * cl);
```

## Arguments

*cl* Pointer to the HSI client

## Description

Return `-errno` on failure, 0 on success

## Name

`hsi_port_unregister_clients` — Unregister an HSI port

## Synopsis

```
void hsi_port_unregister_clients (struct hsi_port * port);
```

## Arguments

*port*    The HSI port to unregister

## Name

`hsi_unregister_controller` — Unregister an HSI controller

## Synopsis

```
void hsi_unregister_controller (struct hsi_controller * hsi);
```

## Arguments

*hsi*    The HSI controller to register

## Name

`hsi_register_controller` — Register an HSI controller and its ports

## Synopsis

```
int hsi_register_controller (struct hsi_controller * hsi);
```

## Arguments

*hsi*    The HSI controller to register

## Description

Returns `-errno` on failure, 0 on success.



## Name

`hsi_register_client_driver` — Register an HSI client to the HSI bus

## Synopsis

```
int hsi_register_client_driver (struct hsi_client_driver * drv);
```

## Arguments

*drv*    HSI client driver to register

## Description

Returns `-errno` on failure, 0 on success.

## Name

`hsi_put_controller` — Free an HSI controller

## Synopsis

```
void hsi_put_controller (struct hsi_controller * hsi);
```

## Arguments

*hsi*    Pointer to the HSI controller to freed

## Description

HSI controller drivers should only use this function if they need to free their allocated `hsi_controller` structures before a successful call to `hsi_register_controller`. Other use is not allowed.

## Name

`hsi_alloc_controller` — Allocate an HSI controller and its ports

## Synopsis

```
struct hsi_controller * hsi_alloc_controller (unsigned int n_ports,  
gfp_t flags);
```

## Arguments

*n\_ports*    Number of ports on the HSI controller

*flags*      Kernel allocation flags

## Description

Return NULL on failure or a pointer to an `hsi_controller` on success.

## Name

`hsi_free_msg` — Free an HSI message

## Synopsis

```
void hsi_free_msg (struct hsi_msg * msg);
```

## Arguments

*msg* Pointer to the HSI message

## Description

Client is responsible to free the buffers pointed by the scatterlists.

## Name

`hsi_alloc_msg` — Allocate an HSI message

## Synopsis

```
struct hsi_msg * hsi_alloc_msg (unsigned int nents, gfp_t flags);
```

## Arguments

*nents*    Number of memory entries

*flags*    Kernel allocation flags

## Description

*nents* can be 0. This mainly makes sense for read transfer. In that case, HSI drivers will call the complete callback when there is data to be read without consuming it.

Return NULL on failure or a pointer to an `hsi_msg` on success.

## Name

`hsi_async` — Submit an HSI transfer to the controller

## Synopsis

```
int hsi_async (struct hsi_client * cl, struct hsi_msg * msg);
```

## Arguments

*cl*     HSI client sending the transfer

*msg*   The HSI transfer passed to controller

## Description

The HSI message must have the channel, ttype, complete and destructor fields set beforehand. If nents > 0 then the client has to initialize also the scatterlists to point to the buffers to write to or read from.

HSI controllers relay on pre-allocated buffers from their clients and they do not allocate buffers on their own.

Once the HSI message transfer finishes, the HSI controller calls the complete callback with the status and actual\_len fields of the HSI message updated. The complete callback can be called before returning from `hsi_async`.

Returns -errno on failure or 0 on success

## Name

`hsi_claim_port` — Claim the HSI client's port

## Synopsis

```
int hsi_claim_port (struct hsi_client * cl, unsigned int share);
```

## Arguments

*cl*        HSI client that wants to claim its port

*share*    Flag to indicate if the client wants to share the port or not.

## Description

Returns -errno on failure, 0 on success.

## Name

`hsi_release_port` — Release the HSI client's port

## Synopsis

```
void hsi_release_port (struct hsi_client * cl);
```

## Arguments

*cl* HSI client which previously claimed its port



## Name

`hsi_register_port_event` — Register a client to receive port events

## Synopsis

```
int hsi_register_port_event (struct hsi_client * cl, void (*handler)  
(struct hsi_client *, unsigned long));
```

## Arguments

*cl*            HSI client that wants to receive port events

*handler*    Event handler callback

## Description

Clients should register a callback to be able to receive events from the ports. Registration should happen after claiming the port. The handler can be called in interrupt context.

Returns -errno on error, or 0 on success.

## Name

`hsi_unregister_port_event` — Stop receiving port events for a client

## Synopsis

```
int hsi_unregister_port_event (struct hsi_client * cl);
```

## Arguments

*cl* HSI client that wants to stop receiving port events

## Description

Clients should call this function before releasing their associated port.

Returns `-errno` on error, or 0 on success.

## Name

`hsi_event` — Notifies clients about port events

## Synopsis

```
int hsi_event (struct hsi_port * port, unsigned long event);
```

## Arguments

*port*     Port where the event occurred

*event*   The event type

## Description

Clients should not be concerned about wake line behavior. However, due to a race condition in HSI HW protocol, clients need to be notified about wake line changes, so they can implement a workaround for it.

## Events

HSI\_EVENT\_START\_RX - Incoming wake line high  
HSI\_EVENT\_STOP\_RX - Incoming wake line down

Returns -errno on error, or 0 on success.

## Name

`hsi_get_channel_id_by_name` — acquire channel id by channel name

## Synopsis

```
int hsi_get_channel_id_by_name (struct hsi_client * cl, char * name);
```

## Arguments

*cl*     HSI client, which uses the channel

*name*   name the channel is known under

## Description

Clients can call this function to get the hsi channel ids similar to requesting IRQs or GPIOs by name. This function assumes the same channel configuration is used for RX and TX.

Returns -errno on error or channel id on success.

---

# Chapter 13. Pulse-Width Modulation (PWM)

Pulse-width modulation is a modulation technique primarily used to control power supplied to electrical devices.

The PWM framework provides an abstraction for providers and consumers of PWM signals. A controller that provides one or more PWM signals is registered as struct `pwm_chip`. Providers are expected to embed this structure in a driver-specific structure. This structure contains fields that describe a particular chip.

A chip exposes one or more PWM signal sources, each of which exposed as a struct `pwm_device`. Operations can be performed on PWM devices to control the period, duty cycle, polarity and active state of the signal.

Note that PWM devices are exclusive resources: they can always only be used by one consumer at a time.

## Name

enum pwm\_polarity — polarity of a PWM signal

## Synopsis

```
enum pwm_polarity {  
    PWM_POLARITY_NORMAL,  
    PWM_POLARITY_INVERSED  
};
```

## Constants

PWM\_POLARITY\_NORMAL    a high signal for the duration of the duty- cycle, followed by a low signal for the remainder of the pulse period

PWM\_POLARITY\_INVERSED a low signal for the duration of the duty- cycle, followed by a high signal for the remainder of the pulse period

## Name

struct pwm\_device — PWM channel object

## Synopsis

```
struct pwm_device {
    const char * label;
    unsigned long flags;
    unsigned int hwpwm;
    unsigned int pwm;
    struct pwm_chip * chip;
    void * chip_data;
    struct mutex lock;
    unsigned int period;
    unsigned int duty_cycle;
    enum pwm_polarity polarity;
};
```

## Members

label	name of the PWM device
flags	flags associated with the PWM device
hwpwm	per-chip relative index of the PWM device
pwm	global index of the PWM device
chip	PWM chip providing this PWM device
chip_data	chip-private data associated with the PWM device
lock	used to serialize accesses to the PWM device where necessary
period	period of the PWM signal (in nanoseconds)
duty_cycle	duty cycle of the PWM signal (in nanoseconds)
polarity	polarity of the PWM signal

## Name

struct pwm\_ops — PWM controller operations

## Synopsis

```
struct pwm_ops {
    int (* request) (struct pwm_chip *chip, struct pwm_device *pwm);
    void (* free) (struct pwm_chip *chip, struct pwm_device *pwm);
    int (* config) (struct pwm_chip *chip, struct pwm_device *pwm, int duty_ns, int p
    int (* set_polarity) (struct pwm_chip *chip, struct pwm_device *pwm, enum pwm_pol
    int (* enable) (struct pwm_chip *chip, struct pwm_device *pwm);
    void (* disable) (struct pwm_chip *chip, struct pwm_device *pwm);
#ifdef CONFIG_DEBUG_FS
    void (* dbg_show) (struct pwm_chip *chip, struct seq_file *s);
#endif
    struct module * owner;
};
```

## Members

request	optional hook for requesting a PWM
free	optional hook for freeing a PWM
config	configure duty cycles and period length for this PWM
set_polarity	configure the polarity of this PWM
enable	enable PWM output toggling
disable	disable PWM output toggling
dbg_show	optional routine to show contents in debugfs
owner	helps prevent removal of modules exporting active PWMs



## Name

struct pwm\_chip — abstract a PWM controller

## Synopsis

```
struct pwm_chip {
    struct device * dev;
    struct list_head list;
    const struct pwm_ops * ops;
    int base;
    unsigned int npwm;
    struct pwm_device * pwms;
    struct pwm_device * (* of_xlate) (struct pwm_chip *pc, const struct of_phandle_arg *args);
    unsigned int of_pwm_n_cells;
    bool can_sleep;
};
```

## Members

dev	device providing the PWMs
list	list node for internal use
ops	callbacks for this PWM controller
base	number of first PWM controlled by this chip
npwm	number of PWMs controlled by this chip
pwms	array of PWM devices allocated by the framework
of_xlate	request a PWM device given a device tree PWM specifier
of_pwm_n_cells	number of cells expected in the device tree PWM specifier
can_sleep	must be true if the .config, .enable or .disable operations may sleep

## Name

`pwm_set_chip_data` — set private chip data for a PWM

## Synopsis

```
int pwm_set_chip_data (struct pwm_device * pwm, void * data);
```

## Arguments

*pwm*    PWM device

*data*   pointer to chip-specific data

## Returns

0 on success or a negative error code on failure.

## Name

`pwm_get_chip_data` — get private chip data for a PWM

## Synopsis

```
void * pwm_get_chip_data (struct pwm_device * pwm);
```

## Arguments

*pwm* PWM device

## Returns

A pointer to the chip-private data for the PWM device.

## Name

`pwmchip_add_with_polarity` — register a new PWM chip

## Synopsis

```
int pwmchip_add_with_polarity (struct pwm_chip * chip, enum pwm_polarity
polarity);
```

## Arguments

*chip*            the PWM chip to add

*polarity*       initial polarity of PWM channels

## Description

Register a new PWM chip. If `chip->base < 0` then a dynamically assigned base will be used. The initial polarity for all channels is specified by the *polarity* parameter.

## Returns

0 on success or a negative error code on failure.

## Name

`pwmchip_add` — register a new PWM chip

## Synopsis

```
int pwmchip_add (struct pwm_chip * chip);
```

## Arguments

*chip* the PWM chip to add

## Description

Register a new PWM chip. If `chip->base < 0` then a dynamically assigned base will be used. The initial polarity for all channels is normal.

## Returns

0 on success or a negative error code on failure.

## Name

`pwmchip_remove` — remove a PWM chip

## Synopsis

```
int pwmchip_remove (struct pwm_chip * chip);
```

## Arguments

*chip* the PWM chip to remove

## Description

Removes a PWM chip. This function may return busy if the PWM chip provides a PWM device that is still requested.

## Returns

0 on success or a negative error code on failure.

## Name

`pwm_request` — request a PWM device

## Synopsis

```
struct pwm_device * pwm_request (int pwm, const char * label);
```

## Arguments

*pwm*      global PWM device index

*label*    PWM device label

## Description

This function is deprecated, use `pwm_get` instead.

## Returns

A pointer to a PWM device or an `ERR_PTR`-encoded error code on failure.

## Name

`pwm_request_from_chip` — request a PWM device relative to a PWM chip

## Synopsis

```
struct pwm_device * pwm_request_from_chip (struct pwm_chip * chip,  
unsigned int index, const char * label);
```

## Arguments

*chip*    PWM chip

*index*   per-chip index of the PWM to request

*label*   a literal description string of this PWM

## Returns

A pointer to the PWM device at the given index of the given PWM chip. A negative error code is returned if the index is not valid for the specified PWM chip or if the PWM device cannot be requested.



## Name

`pwm_free` — free a PWM device

## Synopsis

```
void pwm_free (struct pwm_device * pwm);
```

## Arguments

*pwm* PWM device

## Description

This function is deprecated, use `pwm_put` instead.

## Name

`pwm_config` — change a PWM device configuration

## Synopsis

```
int pwm_config (struct pwm_device * pwm, int duty_ns, int period_ns);
```

## Arguments

<i>pwm</i>	PWM device
<i>duty_ns</i>	"on" time (in nanoseconds)
<i>period_ns</i>	duration (in nanoseconds) of one cycle

## Returns

0 on success or a negative error code on failure.

## Name

`pwm_set_polarity` — configure the polarity of a PWM signal

## Synopsis

```
int pwm_set_polarity (struct pwm_device * pwm, enum pwm_polarity
polarity);
```

## Arguments

*pwm*            PWM device

*polarity*    new polarity of the PWM signal

## Description

Note that the polarity cannot be configured while the PWM device is enabled.

## Returns

0 on success or a negative error code on failure.

## Name

`pwm_enable` — start a PWM output toggling

## Synopsis

```
int pwm_enable (struct pwm_device * pwm);
```

## Arguments

*pwm*   PWM device

## Returns

0 on success or a negative error code on failure.

## Name

`pwm_disable` — stop a PWM output toggling

## Synopsis

```
void pwm_disable (struct pwm_device * pwm);
```

## Arguments

*pwm* PWM device

## Name

`of_pwm_get` — request a PWM via the PWM framework

## Synopsis

```
struct pwm_device * of_pwm_get (struct device_node * np, const char
* con_id);
```

## Arguments

*np*            device node to get the PWM from

*con\_id*       consumer name

## Description

Returns the PWM device parsed from the phandle and index specified in the “pwms” property of a device tree node or a negative error-code on failure. Values parsed from the device tree are stored in the returned PWM device object.

If *con\_id* is NULL, the first PWM device listed in the “pwms” property will be requested. Otherwise the “pwm-names” property is used to do a reverse lookup of the PWM index. This also means that the “pwm-names” property becomes mandatory for devices that look up the PWM device via the *con\_id* parameter.

## Returns

A pointer to the requested PWM device or an `ERR_PTR`-encoded error code on failure.

## Name

`pwm_get` — look up and request a PWM device

## Synopsis

```
struct pwm_device * pwm_get (struct device * dev, const char * con_id);
```

## Arguments

*dev*        device for PWM consumer

*con\_id*    consumer name

## Description

Lookup is first attempted using DT. If the device was not instantiated from a device tree, a PWM chip and a relative index is looked up via a table supplied by board setup code (see `pwm_add_table`).

Once a PWM chip has been found the specified PWM device will be requested and is ready to be used.

## Returns

A pointer to the requested PWM device or an `ERR_PTR`-encoded error code on failure.

## Name

`pwm_put` — release a PWM device

## Synopsis

```
void pwm_put (struct pwm_device * pwm);
```

## Arguments

*pwm* PWM device



## Name

`devm_pwm_get` — resource managed `pwm_get`

## Synopsis

```
struct pwm_device * devm_pwm_get (struct device * dev, const char *  
con_id);
```

## Arguments

*dev*        device for PWM consumer

*con\_id*    consumer name

## Description

This function performs like `pwm_get` but the acquired PWM device will automatically be released on driver detach.

## Returns

A pointer to the requested PWM device or an `ERR_PTR`-encoded error code on failure.

## Name

`devm_of_pwm_get` — resource managed `of_pwm_get`

## Synopsis

```
struct pwm_device * devm_of_pwm_get (struct device * dev, struct  
device_node * np, const char * con_id);
```

## Arguments

*dev*        device for PWM consumer

*np*        device node to get the PWM from

*con\_id*    consumer name

## Description

This function performs like `of_pwm_get` but the acquired PWM device will automatically be released on driver detach.

## Returns

A pointer to the requested PWM device or an `ERR_PTR`-encoded error code on failure.

## Name

devm\_pwm\_put — resource managed pwm\_put

## Synopsis

```
void devm_pwm_put (struct device * dev, struct pwm_device * pwm);
```

## Arguments

*dev*    device for PWM consumer

*pwm*    PWM device

## Description

Release a PWM previously allocated using `devm_pwm_get`. Calling this function is usually not needed because devm-allocated resources are automatically released on driver detach.

## Name

`pwm_can_sleep` — report whether PWM access will sleep

## Synopsis

```
bool pwm_can_sleep (struct pwm_device * pwm);
```

## Arguments

*pwm* PWM device

## Returns

True if accessing the PWM can sleep, false otherwise.