The kernel driver API manual

Release 4.16.0-rc4+

The kernel development community

March 08, 2018

The kernel offers a wide variety of interfaces to support the development of device drivers. This document is an only somewhat organized collection of some of those interfaces — it will hopefully get better over time! The available subsections can be seen below.

Table of contents

CHAPTER

ONE

DRIVER BASICS

Driver Entry and Exit points

module_init(*x*)
driver initialization entry point

Parameters

x function to be run at kernel boot time or module insertion

Description

module_init() will either be called during `do_initcalls()` (if builtin) or at module insertion time (if a module). There can only be one per module.

module_exit(*x*)
driver exit entry point

Parameters

x function to be run when driver is removed

Description

module_exit() will wrap the driver clean-up code with `cleanup_module()` when used with rmmod when the driver is a module. If the driver is statically compiled into the kernel, *module_exit()* has no effect. There can only be one per module.

Driver device table

struct **usb_device_id**
identifies USB devices for probing and hotplugging

Definition

```
struct usb_device_id {
  __u16 match_flags;
  __u16 idVendor;
  __u16 idProduct;
  __u16 bcdDevice_lo;
  __u16 bcdDevice_hi;
  __u8 bDeviceClass;
  __u8 bDeviceSubClass;
  __u8 bDeviceProtocol;
  __u8 bInterfaceClass;
  __u8 bInterfaceSubClass;
  __u8 bInterfaceProtocol;
  __u8 bInterfaceNumber;
  kernel_ulong_t driver_info ;
};
```

Members

match_flags Bit mask controlling which of the other fields are used to match against new devices. Any field except for driver_info may be used, although some only make sense in conjunction with other fields. This is usually set by a USB_DEVICE_*() macro, which sets all other fields in this structure except for driver_info.

idVendor USB vendor ID for a device; numbers are assigned by the USB forum to its members.

idProduct Vendor-assigned product ID.

bcdDevice_lo Low end of range of vendor-assigned product version numbers. This is also used to identify individual product versions, for a range consisting of a single device.

bcdDevice_hi High end of version number range. The range of product versions is inclusive.

bDeviceClass Class of device; numbers are assigned by the USB forum. Products may choose to implement classes, or be vendor-specific. Device classes specify behavior of all the interfaces on a device.

bDeviceSubClass Subclass of device; associated with bDeviceClass.

bDeviceProtocol Protocol of device; associated with bDeviceClass.

bInterfaceClass Class of interface; numbers are assigned by the USB forum. Products may choose to implement classes, or be vendor-specific. Interface classes specify behavior only of a given interface; other interfaces may support other classes.

bInterfaceSubClass Subclass of interface; associated with bInterfaceClass.

bInterfaceProtocol Protocol of interface; associated with bInterfaceClass.

bInterfaceNumber Number of interface; composite devices may use fixed interface numbers to differentiate between vendor-specific interfaces.

driver_info Holds information used by the driver. Usually it holds a pointer to a descriptor understood by the driver, or perhaps device flags.

Description

In most cases, drivers will create a table of device IDs by using *USB_DEVICE()*, or similar macros designed for that purpose. They will then export it to userspace using MODULE_DEVICE_TABLE(), and provide it to the USB core through their usb_driver structure.

See the *usb_match_id()* function for information about how matches are performed. Briefly, you will normally use one of several macros to help construct these entries. Each entry you provide will either identify one or more specific products, or will identify a class of products which have agreed to behave the same. You should put the more specific matches towards the beginning of your table, so that driver_info can record quirks of specific products.

struct **mdio_device_id**
identifies PHY devices on an MDIO/MII bus

Definition

```
struct mdio_device_id {
  __u32 phy_id;
  __u32 phy_id_mask;
};
```

Members

phy_id The result of (mdio_read(MII_PHYSID1) << 16 | mdio_read(PHYSID2)) & **phy_id_mask** for this PHY type

phy_id_mask Defines the significant bits of **phy_id**. A value of 0 is used to terminate an array of struct mdio_device_id.

struct **amba_id**
identifies a device on an AMBA bus

Definition

```
struct amba_id {
  unsigned int            id;
  unsigned int            mask;
  void *data;
};
```

Members

id The significant bits if the hardware device ID

mask Bitmask specifying which bits of the id field are significant when matching. A driver binds to a device when ((hardware device ID) & mask) == id.

data Private data used by the driver.

struct **mips_cdmm_device_id**
identifies devices in MIPS CDMM bus

Definition

```
struct mips_cdmm_device_id {
  __u8 type;
};
```

Members

type Device type identifier.

struct **mei_cl_device_id**
MEI client device identifier

Definition

```
struct mei_cl_device_id {
  char name[MEI_CL_NAME_SIZE];
  uuid_le uuid;
  __u8 version;
  kernel_ulong_t driver_info;
};
```

Members

name helper name

uuid client uuid

version client protocol version

driver_info information used by the driver.

Description

identifies mei client device by uuid and name

struct **rio_device_id**
RIO device identifier

Definition

```
struct rio_device_id {
  __u16 did, vid;
  __u16 asm_did, asm_vid;
};
```

Members

did RapidIO device ID

vid RapidIO vendor ID

asm_did RapidIO assembly device ID

asm_vid RapidIO assembly vendor ID

Description

Identifies a RapidIO device based on both the device/vendor IDs and the assembly device/vendor IDs.

struct **fsl_mc_device_id**
MC object device identifier

Definition

```
struct fsl_mc_device_id {
  __u16 vendor;
  const char obj_type[16];
};
```

Members

vendor vendor ID

obj_type MC object type

Description

Type of entries in the "device Id" table for MC object devices supported by a MC object device driver. The last entry of the table has vendor set to 0x0

struct **tb_service_id**
Thunderbolt service identifiers

Definition

```
struct tb_service_id {
  __u32 match_flags;
  char protocol_key[8 + 1];
  __u32 protocol_id;
  __u32 protocol_version;
  __u32 protocol_revision;
  kernel_ulong_t driver_data;
};
```

Members

match_flags Flags used to match the structure

protocol_key Protocol key the service supports

protocol_id Protocol id the service supports

protocol_version Version of the protocol

protocol_revision Revision of the protocol software

driver_data Driver specific data

Description

Thunderbolt XDomain services are exposed as devices where each device carries the protocol information the service supports. Thunderbolt XDomain service drivers match against that information.

Delaying, scheduling, and timer routines

struct **prev_cputime**
snapshot of system and user cputime

Definition

```
struct prev_cputime {
#ifndef CONFIG_VIRT_CPU_ACCOUNTING_NATIVE;
  u64 utime;
  u64 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.

struct **task_cputime**
collected CPU time counts

Definition

```
struct task_cputime {
  u64 utime;
  u64 stime;
  unsigned long long              sum_exec_runtime;
};
```

Members

utime time spent in user mode, in nanoseconds

stime time spent in kernel mode, in nanoseconds

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.

int **pid_alive**(const struct task_struct * *p*)
check that a task structure is not stale

Parameters

const struct task_struct * 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.

int **is_global_init**(struct task_struct * *tsk*)
check if a task structure is init. Since init is free to have sub-threads we need to check tgid.

Parameters

`struct task_struct * 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.

int **`task_nice`**(const struct task_struct * *p*)
    return the nice value of a given task.

Parameters

`const struct task_struct * p` the task in question.

Return

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

bool **`is_idle_task`**(const struct task_struct * *p*)
    is the specified task an idle task?

Parameters

`const struct task_struct * p` the task in question.

Return

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

int **`wake_up_process`**(struct task_struct * *p*)
    Wake up a specific process

Parameters

`struct task_struct * 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.

void **`preempt_notifier_register`**(struct preempt_notifier * *notifier*)
    tell me when current is being preempted & rescheduled

Parameters

`struct preempt_notifier * notifier` notifier struct to register

void **`preempt_notifier_unregister`**(struct preempt_notifier * *notifier*)
    no longer interested in preemption notifications

Parameters

`struct preempt_notifier * notifier` notifier struct to unregister

Description

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

__visible void __sched notrace **`preempt_schedule_notrace`**(void)
    preempt_schedule called by tracing

Parameters

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.

int **sched_setscheduler**(struct task_struct * *p*, int *policy*, const struct sched_param * *param*)
change the scheduling policy and/or RT priority of a thread.

Parameters

struct task_struct * p the task in question.

int policy new policy.

const struct sched_param * param structure containing the new RT priority.

Return

0 on success. An error code otherwise.

NOTE that the task may be already dead.

int **sched_setscheduler_nocheck**(struct task_struct * *p*, int *policy*, const struct sched_param * *param*)
change the scheduling policy and/or RT priority of a thread from kernelspace.

Parameters

struct task_struct * p the task in question.

int policy new policy.

const struct sched_param * 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.

void __sched **yield**(void)
yield the current processor to other threads.

Parameters

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!

int __sched **yield_to**(struct task_struct * *p*, bool *preempt*)
yield the current processor to another thread in your thread group, or accelerate that thread toward the processor it's on.

Parameters

struct task_struct * p target task

bool 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.

int **cpupri_find**(struct cpupri * *cp*, struct task_struct * *p*, struct cpumask * *lowest_mask*)
find the best (lowest-pri) CPU in the system

Parameters

struct cpupri * cp The cpupri context

struct task_struct * p The task

struct cpumask * lowest_mask 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

void **cpupri_set**(struct cpupri * *cp*, int *cpu*, int *newpri*)
update the cpu priority setting

Parameters

struct cpupri * cp The cpupri context

int cpu The target cpu

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

Note

Assumes cpu_rq(cpu)->lock is locked

Return

(void)

int **cpupri_init**(struct cpupri * *cp*)
initialize the cpupri structure

Parameters

struct cpupri * cp The cpupri context

Return

-ENOMEM on memory allocation failure.

void **`cpupri_cleanup`**(struct cpupri * *cp*)
clean up the cpupri structure

Parameters

`struct cpupri * cp` The cpupri context

void **`update_tg_load_avg`**(struct cfs_rq * *cfs_rq*, int *force*)
update the tg's load avg

Parameters

`struct cfs_rq * cfs_rq` the cfs_rq whose avg changed

`int force` update regardless of how small the difference

Description

This function 'ensures': tg->load_avg := Sum tg->cfs_rq[]->avg.load. However, because tg->load_avg is a global value there are performance considerations.

In order to avoid having to look at the other cfs_rq's, we use a differential update where we store the last value we propagated. This in turn allows skipping updates if the differential is 'small'.

Updating tg's load_avg is necessary before `update_cfs_share()`.

int **`update_cfs_rq_load_avg`**(u64 *now*, struct cfs_rq * *cfs_rq*)
update the cfs_rq's load/util averages

Parameters

`u64 now` current time, as per `cfs_rq_clock_task()`

`struct cfs_rq * cfs_rq` cfs_rq to update

Description

The cfs_rq avg is the direct sum of all its entities (blocked and runnable) avg. The immediate corollary is that all (fair) tasks must be attached, see `post_init_entity_util_avg()`.

cfs_rq->avg is used for `task_h_load()` and `update_cfs_share()` for example.

Returns true if the load decayed or we removed load.

Since both these conditions indicate a changed cfs_rq->avg.load we should call *`update_tg_load_avg()`* when this function returns true.

void **`attach_entity_load_avg`**(struct cfs_rq * *cfs_rq*, struct sched_entity * *se*)
attach this entity to its cfs_rq load avg

Parameters

`struct cfs_rq * cfs_rq` cfs_rq to attach to

`struct sched_entity * se` sched_entity to attach

Description

Must call *`update_cfs_rq_load_avg()`* before this, since we rely on cfs_rq->avg.last_update_time being current.

void **`detach_entity_load_avg`**(struct cfs_rq * *cfs_rq*, struct sched_entity * *se*)
detach this entity from its cfs_rq load avg

Parameters

`struct cfs_rq * cfs_rq` cfs_rq to detach from

`struct sched_entity * se` sched_entity to detach

Description

Must call *`update_cfs_rq_load_avg()`* before this, since we rely on cfs_rq->avg.last_update_time being current.

void **`cpu_load_update`**(struct rq * *this_rq*, unsigned long *this_load*, unsigned long *pending_updates*)
update the rq->cpu_load[] statistics

Parameters

`struct rq * this_rq` The rq to update statistics for

`unsigned long this_load` The current load

`unsigned long pending_updates` The number of missed updates

Description

Update rq->cpu_load[] statistics. This function is usually called every scheduler tick (TICK_NSEC).

This function computes a decaying average:

load[i]' = (1 - 1/2^i) * load[i] + (1/2^i) * load

Because of NOHZ it might not get called on every tick which gives need for the **pending_updates** argument.

load[i]_n = (1 - 1/2^i) * load[i]_n-1 + (1/2^i) * load_n-1 = A * load[i]_n-1 + B ; A := (1 - 1/2^i), B := (1/2^i) * load = A * (A * load[i]_n-2 + B) + B = A * (A * (A * load[i]_n-3 + B) + B) + B = A^3 * load[i]_n-3 + (A^2 + A + 1) * B = A^n * load[i]_0 + (A^(n-1) + A^(n-2) + ... + 1) * B = A^n * load[i]_0 + ((1 - A^n) / (1 - A)) * B = (1 - 1/2^i)^n * (load[i]_0 - load) + load

In the above we've assumed load_n := load, which is true for NOHZ_FULL as any change in load would have resulted in the tick being turned back on.

For regular NOHZ, this reduces to:

load[i]_n = (1 - 1/2^i)^n * load[i]_0

see `decay_load_misses()`. For NOHZ_FULL we get to subtract and add the extra term.

int **`get_sd_load_idx`**(struct sched_domain * *sd*, enum cpu_idle_type *idle*)
Obtain the load index for a given sched domain.

Parameters

`struct sched_domain * sd` The sched_domain whose load_idx is to be obtained.

`enum cpu_idle_type idle` The idle status of the CPU for whose sd load_idx is obtained.

Return

The load index.

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*)
Update sched_group's statistics for load balancing.

Parameters

`struct lb_env * env` The load balancing environment.

`struct sched_group * group` sched_group whose statistics are to be updated.

`int load_idx` Load index of sched_domain of this_cpu for load calc.

`int local_group` Does group contain this_cpu.

`struct sg_lb_stats * sgs` variable to hold the statistics for this group.

`bool * overload` Indicate more than one runnable task for any CPU.

bool **`update_sd_pick_busiest`**(struct lb_env * *env*, struct sd_lb_stats * *sds*, struct sched_group * *sg*, struct sg_lb_stats * *sgs*)
return 1 on busiest group

Parameters

struct lb_env * env The load balancing environment.

struct sd_lb_stats * sds sched_domain statistics

struct sched_group * sg sched_group candidate to be checked for being the busiest

struct sg_lb_stats * 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.

void **update_sd_lb_stats**(struct lb_env * *env*, struct sd_lb_stats * *sds*)
Update sched_domain's statistics for load balancing.

Parameters

struct lb_env * env The load balancing environment.

struct sd_lb_stats * sds variable to hold the statistics for this sched_domain.

int **check_asym_packing**(struct lb_env * *env*, struct sd_lb_stats * *sds*)
Check to see if the group is packed into the sched domain.

Parameters

struct lb_env * env The load balancing environment.

struct sd_lb_stats * sds Statistics of the sched_domain which is to be packed

Description

This is primarily intended to 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 env->imbalance.

void **fix_small_imbalance**(struct lb_env * *env*, struct sd_lb_stats * *sds*)
Calculate the minor imbalance that exists amongst the groups of a sched_domain, during load balancing.

Parameters

struct lb_env * env The load balancing environment.

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

void **calculate_imbalance**(struct lb_env * *env*, struct sd_lb_stats * *sds*)
Calculate the amount of imbalance present within the groups of a given sched_domain during load balance.

Parameters

struct lb_env * env load balance environment

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

struct sched_group * **find_busiest_group**(struct lb_env * *env*)
Returns the busiest group within the sched_domain if there is an imbalance.

Parameters

struct lb_env * 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.

DECLARE_COMPLETION(*work*)
    declare and initialize a completion structure

Parameters

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.

DECLARE_COMPLETION_ONSTACK(*work*)
    declare and initialize a completion structure

Parameters

work identifier for the completion structure

Description

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

void **__init_completion**(struct completion * *x*)
    Initialize a dynamically allocated completion

Parameters

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

Description

This inline function will initialize a dynamically created completion structure.

void **reinit_completion**(struct completion * *x*)
    reinitialize a completion structure

Parameters

struct completion * 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.

unsigned long **__round_jiffies**(unsigned long *j*, int *cpu*)
    function to round jiffies to a full second

Parameters

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

int 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.

unsigned long **`__round_jiffies_relative`**(unsigned long *j*, int *cpu*)
function to round jiffies to a full second

Parameters

`unsigned long j` the time in (relative) jiffies that should be rounded

`int 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.

unsigned long **`round_jiffies`**(unsigned long *j*)
function to round jiffies to a full second

Parameters

`unsigned long 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.

unsigned long **`round_jiffies_relative`**(unsigned long *j*)
function to round jiffies to a full second

Parameters

`unsigned long 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.

unsigned long **`__round_jiffies_up`**(unsigned long *j*, int *cpu*)
function to round jiffies up to a full second

Parameters

`unsigned long j` the time in (absolute) jiffies that should be rounded

`int 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.

unsigned long **`__round_jiffies_up_relative`**(unsigned long *j*, int *cpu*)
function to round jiffies up to a full second

Parameters

`unsigned long j` the time in (relative) jiffies that should be rounded

`int 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.

unsigned long **`round_jiffies_up`**(unsigned long *j*)
function to round jiffies up to a full second

Parameters

`unsigned long 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.

unsigned long **`round_jiffies_up_relative`**(unsigned long *j*)
function to round jiffies up to a full second

Parameters

`unsigned long 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.

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

Parameters

`struct timer_list * timer` the timer to be initialized

`void (*)(struct timer_list *) func` timer callback function

`unsigned int flags` timer flags

`const char * name` name of the timer

`struct lock_class_key * 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.

int **`mod_timer_pending`**(struct timer_list * *timer*, unsigned long *expires*)
modify a pending timer's timeout

Parameters

`struct timer_list * timer` the pending timer to be modified

unsigned long 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.

int **mod_timer**(struct timer_list * *timer*, unsigned long *expires*)
modify a timer's timeout

Parameters

struct timer_list * timer the timer to be modified

unsigned long 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.)

int **timer_reduce**(struct timer_list * *timer*, unsigned long *expires*)
Modify a timer's timeout if it would reduce the timeout

Parameters

struct timer_list * timer The timer to be modified

unsigned long expires New timeout in jiffies

Description

timer_reduce() is very similar to *mod_timer()*, except that it will only modify a running timer if that would reduce the expiration time (it will start a timer that isn't running).

void **add_timer**(struct timer_list * *timer*)
start a timer

Parameters

struct timer_list * timer the timer to be added

Description

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

The timer's ->expires, ->function fields must be set prior calling this function.

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

void **add_timer_on**(struct timer_list * *timer*, int *cpu*)
start a timer on a particular CPU

Parameters

struct timer_list * timer the timer to be added

int cpu the CPU to start it on

Description

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

int **del_timer**(struct timer_list * *timer*)
deactivate a timer.

Parameters

struct timer_list * 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.)

int **try_to_del_timer_sync**(struct timer_list * *timer*)
Try to deactivate a timer

Parameters

struct timer_list * timer timer to delete

Description

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

int **del_timer_sync**(struct timer_list * *timer*)
deactivate a timer and wait for the handler to finish.

Parameters

struct timer_list * 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.

signed long __sched **schedule_timeout**(signed long *timeout*)
sleep until timeout

Parameters

`signed long 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 unless the current task is explicitly woken up, (e.g. by *`wake_up_process()`*)".

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

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`.

Returns 0 when the timer has expired otherwise the remaining time in jiffies will be returned. In all cases the return value is guaranteed to be non-negative.

void **`msleep`**(unsigned int *msecs*)
sleep safely even with waitqueue interruptions

Parameters

`unsigned int msecs` Time in milliseconds to sleep for

unsigned long **`msleep_interruptible`**(unsigned int *msecs*)
sleep waiting for signals

Parameters

`unsigned int msecs` Time in milliseconds to sleep for

void __sched **`usleep_range`**(unsigned long *min*, unsigned long *max*)
Sleep for an approximate time

Parameters

`unsigned long min` Minimum time in usecs to sleep

`unsigned long max` Maximum time in usecs to sleep

Description

In non-atomic context where the exact wakeup time is flexible, use *`usleep_range()`* instead of `udelay()`. The sleep improves responsiveness by avoiding the CPU-hogging busy-wait of `udelay()`, and the range reduces power usage by allowing hrtimers to take advantage of an already- scheduled interrupt instead of scheduling a new one just for this sleep.

Wait queues and Wake events

int **`waitqueue_active`**(struct wait_queue_head * *wq_head*)

- locklessly test for waiters on the queue

Parameters

`struct wait_queue_head * wq_head` the waitqueue to test for waiters

Description

returns true if the wait list is not empty

NOTE

this function is lockless and requires care, incorrect usage _will_ lead to sporadic and non-obvious failure.

Use either while holding wait_queue_head::lock or when used for wakeups with an extra `smp_mb()` like:

CPU0 - waker CPU1 - waiter

for (;;) {

cond = true; prepare_to_wait(wq_head, `wait`, state); `smp_mb()`; // `smp_mb()` from `set_current_state()` if (waitqueue_active(wq_head)) if (**cond**)

wake_up(wq_head); break;

`schedule();`

} finish_wait(wq_head, `wait`);

Because without the explicit `smp_mb()` it's possible for the *`waitqueue_active()`* load to get hoisted over the **cond** store such that we'll observe an empty wait list while the waiter might not observe **cond**.

Also note that this 'optimization' trades a `spin_lock()` for an `smp_mb()`, which (when the lock is uncontended) are of roughly equal cost.

bool **`wq_has_sleeper`**(struct wait_queue_head * *wq_head*)
check if there are any waiting processes

Parameters

`struct wait_queue_head * wq_head` wait queue head

Description

Returns true if wq_head has waiting processes

Please refer to the comment for waitqueue_active.

`wait_event`(*wq_head*, *condition*)
sleep until a condition gets true

Parameters

`wq_head` 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_head** is woken up.

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

`wait_event_freezable`(*wq_head*, *condition*)
sleep (or freeze) until a condition gets true

Parameters

`wq_head` 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_head** is woken up.

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

`wait_event_timeout`(*wq_head*, *condition*, *timeout*)
sleep until a condition gets true or a timeout elapses

Parameters

wq_head the waitqueue to wait on

condition a C expression for the event to wait for

timeout 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_head** is woken up.

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

Return

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.

wait_event_cmd(*wq_head*, *condition*, *cmd1*, *cmd2*)
sleep until a condition gets true

Parameters

wq_head the waitqueue to wait on

condition a C expression for the event to wait for

cmd1 the command will be executed before sleep

cmd2 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_head** is woken up.

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

wait_event_interruptible(*wq_head*, *condition*)
sleep until a condition gets true

Parameters

wq_head 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_head** 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.

wait_event_interruptible_timeout(*wq_head*, *condition*, *timeout*)
sleep until a condition gets true or a timeout elapses

Parameters

wq_head the waitqueue to wait on

condition a C expression for the event to wait for

timeout 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_head** is woken up.

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

Return

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.

`wait_event_hrtimeout`(*wq_head*, *condition*, *timeout*)
sleep until a condition gets true or a timeout elapses

Parameters

`wq_head` the waitqueue to wait on

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

`timeout` timeout, as a ktime_t

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_head** 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.

`wait_event_interruptible_hrtimeout`(*wq*, *condition*, *timeout*)
sleep until a condition gets true or a timeout elapses

Parameters

`wq` the waitqueue to wait on

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

`timeout` timeout, as a ktime_t

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.

`wait_event_interruptible_locked`(*wq*, *condition*)
sleep until a condition gets true

Parameters

`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.

`wait_event_interruptible_locked_irq`(*wq*, *condition*)
sleep until a condition gets true

Parameters

`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.

`wait_event_interruptible_exclusive_locked`(*wq*, *condition*)
sleep exclusively until a condition gets true

Parameters

`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 awaken 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.

`wait_event_interruptible_exclusive_locked_irq`(*wq*, *condition*)
sleep until a condition gets true

Parameters

`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 awaken 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.

`wait_event_killable`(*wq_head*, *condition*)
sleep until a condition gets true

Parameters

`wq_head` 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_head** 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.

`wait_event_killable_timeout`(*wq_head*, *condition*, *timeout*)
sleep until a condition gets true or a timeout elapses

Parameters

`wq_head` the waitqueue to wait on

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

`timeout` timeout, in jiffies

Description

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

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

Return

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 kill signal.

Only kill signals interrupt this process.

`wait_event_lock_irq_cmd`(*wq_head*, *condition*, *lock*, *cmd*)
sleep until a condition gets true. The condition is checked under the lock. This is expected to be called with the lock taken.

Parameters

`wq_head` 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 cmd and `schedule()` and reacquired afterwards.

`cmd` 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_head** 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.

`wait_event_lock_irq`(*wq_head*, *condition*, *lock*)
sleep until a condition gets true. The condition is checked under the lock. This is expected to be called with the lock taken.

Parameters

`wq_head` 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_head** 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.

`wait_event_interruptible_lock_irq_cmd`(*wq_head*, *condition*, *lock*, *cmd*)
sleep until a condition gets true. The condition is checked under the lock. This is expected to be called with the lock taken.

Parameters

`wq_head` 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 cmd and `schedule()` and reacquired afterwards.

`cmd` 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_head** 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.

`wait_event_interruptible_lock_irq`(*wq_head*, *condition*, *lock*)
sleep until a condition gets true. The condition is checked under the lock. This is expected to be called with the lock taken.

Parameters

`wq_head` 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_head** 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.

`wait_event_interruptible_lock_irq_timeout`(*wq_head*, *condition*, *lock*, *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.

Parameters

`wq_head` 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.

`timeout` 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_head** 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.

void **`__wake_up`**(struct wait_queue_head * *wq_head*, unsigned int *mode*, int *nr_exclusive*, void * *key*)
wake up threads blocked on a waitqueue.

Parameters

`struct wait_queue_head * wq_head` the waitqueue

`unsigned int mode` which threads

`int nr_exclusive` how many wake-one or wake-many threads to wake up

`void * key` 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.

void **`__wake_up_sync_key`**(struct wait_queue_head * *wq_head*, unsigned int *mode*, int *nr_exclusive*, void * *key*)
wake up threads blocked on a waitqueue.

Parameters

`struct wait_queue_head * wq_head` the waitqueue

`unsigned int mode` which threads

`int nr_exclusive` how many wake-one or wake-many threads to wake up

`void * key` 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.

void **finish_wait**(struct wait_queue_head * *wq_head*, struct wait_queue_entry * *wq_entry*)
clean up after waiting in a queue

Parameters

struct wait_queue_head * wq_head waitqueue waited on

struct wait_queue_entry * wq_entry wait descriptor

Description

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

High-resolution timers

ktime_t **ktime_set**(const s64 *secs*, const unsigned long *nsecs*)
Set a ktime_t variable from a seconds/nanoseconds value

Parameters

const s64 secs seconds to set

const unsigned long nsecs nanoseconds to set

Return

The ktime_t representation of the value.

int **ktime_compare**(const ktime_t *cmp1*, const ktime_t *cmp2*)
Compares two ktime_t variables for less, greater or equal

Parameters

const ktime_t cmp1 comparable1

const ktime_t cmp2 comparable2

Return

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

bool **ktime_after**(const ktime_t *cmp1*, const ktime_t *cmp2*)
Compare if a ktime_t value is bigger than another one.

Parameters

const ktime_t cmp1 comparable1

const ktime_t cmp2 comparable2

Return

true if cmp1 happened after cmp2.

bool **ktime_before**(const ktime_t *cmp1*, const ktime_t *cmp2*)
Compare if a ktime_t value is smaller than another one.

Parameters

const ktime_t cmp1 comparable1

const ktime_t cmp2 comparable2

Return

true if cmp1 happened before cmp2.

bool **ktime_to_timespec_cond**(const ktime_t *kt*, struct timespec * *ts*)
convert a ktime_t variable to timespec format only if the variable contains data

Parameters

const ktime_t kt the ktime_t variable to convert

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

Return

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

bool **ktime_to_timespec64_cond**(const ktime_t *kt*, struct timespec64 * *ts*)
convert a ktime_t variable to timespec64 format only if the variable contains data

Parameters

const ktime_t kt the ktime_t variable to convert

struct timespec64 * ts the timespec variable to store the result in

Return

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

struct **hrtimer**
the basic hrtimer structure

Definition

```
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;
  u8 is_soft;
};
```

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

is_soft Set if hrtimer will be expired in soft interrupt context.

Description

The hrtimer structure must be initialized by *hrtimer_init()*

struct **hrtimer_sleeper**
simple sleeper structure

Definition

```
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.

struct **hrtimer_clock_base**
the timer base for a specific clock

Definition

```
struct hrtimer_clock_base {
  struct hrtimer_cpu_base *cpu_base;
  unsigned int            index;
  clockid_t clockid;
  seqcount_t seq;
  struct hrtimer          *running;
  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

seq seqcount around __run_hrtimer

running pointer to the currently running hrtimer

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

struct **hrtimer_cpu_base**
the per cpu clock bases

Definition

```
struct hrtimer_cpu_base {
  raw_spinlock_t lock;
  unsigned int                cpu;
  unsigned int                active_bases;
  unsigned int                clock_was_set_seq;
  unsigned int                hres_active             : 1,in_hrtirq              : 1,hang_detected
#ifdef CONFIG_HIGH_RES_TIMERS;
  unsigned int                nr_events;
  unsigned short              nr_retries;
  unsigned short              nr_hangs;
  unsigned int                max_hang_time;
#endif;
  ktime_t expires_next;
  struct hrtimer                  *next_timer;
  ktime_t softirq_expires_next;
  struct hrtimer                  *softirq_next_timer;
  struct hrtimer_clock_base       clock_base[HRTIMER_MAX_CLOCK_BASES];
};
```

Members

lock lock protecting the base and associated clock bases and timers

cpu cpu number

active_bases Bitfield to mark bases with active timers

clock_was_set_seq Sequence counter of clock was set events

hres_active State of high resolution mode

in_hrtirq `hrtimer_interrupt()` is currently executing

hang_detected The last hrtimer interrupt detected a hang

softirq_activated displays, if the softirq is raised - update of softirq related settings is not required then.

nr_events Total number of hrtimer interrupt events

nr_retries Total number of hrtimer interrupt retries

nr_hangs Total number of hrtimer interrupt hangs

max_hang_time Maximum time spent in hrtimer_interrupt

expires_next absolute time of the next event, is required for remote hrtimer enqueue; it is the total first expiry time (hard and soft hrtimer are taken into account)

next_timer Pointer to the first expiring timer

softirq_expires_next Time to check, if soft queues needs also to be expired

softirq_next_timer Pointer to the first expiring softirq based timer

clock_base array of clock bases for this cpu

Note

next_timer is just an optimization for __remove_hrtimer(). Do not dereference the pointer because it is not reliable on cross cpu removals.

void **hrtimer_start**(struct *hrtimer* * *timer*, ktime_t *tim*, const enum hrtimer_mode *mode*)
(re)start an hrtimer

Parameters

struct hrtimer * timer the timer to be added

ktime_t tim expiry time

const enum hrtimer_mode mode timer mode: absolute (HRTIMER_MODE_ABS) or relative (HRTIMER_MODE_REL), and pinned (HRTIMER_MODE_PINNED); softirq based mode is considered for debug purpose only!

u64 **hrtimer_forward_now**(struct *hrtimer* * *timer*, ktime_t *interval*)
forward the timer expiry so it expires after now

Parameters

struct hrtimer * timer hrtimer to forward

ktime_t 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.

u64 **hrtimer_forward**(struct *hrtimer* * *timer*, ktime_t *now*, ktime_t *interval*)
forward the timer expiry

Parameters

struct hrtimer * timer hrtimer to forward

ktime_t now forward past this time

ktime_t 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.

void **hrtimer_start_range_ns**(struct *hrtimer* * *timer*, ktime_t *tim*, u64 *delta_ns*, const enum hrtimer_mode *mode*)
(re)start an hrtimer

Parameters

struct hrtimer * timer the timer to be added

ktime_t tim expiry time

u64 delta_ns "slack" range for the timer

const enum hrtimer_mode mode timer mode: absolute (HRTIMER_MODE_ABS) or relative (HRTIMER_MODE_REL), and pinned (HRTIMER_MODE_PINNED); softirq based mode is considered for debug purpose only!

int **hrtimer_try_to_cancel**(struct *hrtimer* * *timer*)
try to deactivate a timer

Parameters

struct hrtimer * timer hrtimer to stop

Return

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

-1 when the timer is currently executing the callback function and cannot be stopped

int **hrtimer_cancel**(struct *hrtimer* * *timer*)
cancel a timer and wait for the handler to finish.

Parameters

struct hrtimer * timer the timer to be cancelled

Return

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

ktime_t **`__hrtimer_get_remaining`**(const struct *hrtimer* * *timer*, bool *adjust*)
get remaining time for the timer

Parameters

`const struct hrtimer * timer` the timer to read

`bool adjust` adjust relative timers when CONFIG_TIME_LOW_RES=y

void **`hrtimer_init`**(struct *hrtimer* * *timer*, clockid_t *clock_id*, enum hrtimer_mode *mode*)
initialize a timer to the given clock

Parameters

`struct hrtimer * timer` the timer to be initialized

`clockid_t clock_id` the clock to be used

`enum hrtimer_mode mode` The modes which are relevant for intitialization: HRTIMER_MODE_ABS, HRTIMER_MODE_REL, HRTIMER_MODE_ABS_SOFT, HRTIMER_MODE_REL_SOFT

Description

The PINNED variants of the above can be handed in, but the PINNED bit is ignored as pinning happens when the hrtimer is started

int __sched **`schedule_hrtimeout_range`**(ktime_t * *expires*, u64 *delta*, const enum hrtimer_mode *mode*)
sleep until timeout

Parameters

`ktime_t * expires` timeout value (ktime_t)

`u64 delta` slack in expires timeout (ktime_t)

`const enum hrtimer_mode mode` timer mode

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 unless the current task is explicitly woken up, (e.g. by *`wake_up_process()`*).

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

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

Returns 0 when the timer has expired. If the task was woken before the timer expired by a signal (only possible in state TASK_INTERRUPTIBLE) or by an explicit wakeup, it returns -EINTR.

int __sched **`schedule_hrtimeout`**(ktime_t * *expires*, const enum hrtimer_mode *mode*)
sleep until timeout

Parameters

`ktime_t * expires` timeout value (ktime_t)

`const enum hrtimer_mode mode` timer mode

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 unless the current task is explicitly woken up, (e.g. by *wake_up_process()*).

TASK_INTERRUPTIBLE - the routine may return early if a signal is delivered to the current task or the current task is explicitly woken up.

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

Returns 0 when the timer has expired. If the task was woken before the timer expired by a signal (only possible in state TASK_INTERRUPTIBLE) or by an explicit wakeup, it returns -EINTR.

Workqueues and Kevents

struct **workqueue_attrs**
A struct for workqueue attributes.

Definition

```
struct workqueue_attrs {
  int nice;
  cpumask_var_t cpumask;
  bool no_numa;
};
```

Members

nice nice level

cpumask allowed CPUs

no_numa disable NUMA affinity

Unlike other fields, no_numa isn't a property of a worker_pool. It only modifies how *apply_workqueue_attrs()* select pools and thus doesn't participate in pool hash calculations or equality comparisons.

Description

This can be used to change attributes of an unbound workqueue.

work_pending(*work*)
Find out whether a work item is currently pending

Parameters

work The work item in question

delayed_work_pending(*w*)
Find out whether a delayable work item is currently pending

Parameters

w The work item in question

alloc_workqueue(*fmt*, *flags*, *max_active*, *args...*)
allocate a workqueue

Parameters

fmt printf format for the name of the workqueue

flags WQ_* flags

max_active max in-flight work items, 0 for default

args... args for **fmt**

Description

Allocate a workqueue with the specified parameters. For detailed information on WQ_* flags, please refer to Documentation/core-api/workqueue.rst.

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.

Return

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

alloc_ordered_workqueue(*fmt*, *flags*, *args...*)
allocate an ordered workqueue

Parameters

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**

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.

Return

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

bool **queue_work**(struct workqueue_struct * *wq*, struct work_struct * *work*)
queue work on a workqueue

Parameters

struct workqueue_struct * wq workqueue to use

struct work_struct * 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.

bool **queue_delayed_work**(struct workqueue_struct * *wq*, struct delayed_work * *dwork*, unsigned long *delay*)
queue work on a workqueue after delay

Parameters

struct workqueue_struct * wq workqueue to use

struct delayed_work * dwork delayable work to queue

unsigned long delay number of jiffies to wait before queueing

Description

Equivalent to *queue_delayed_work_on()* but tries to use the local CPU.

bool **mod_delayed_work**(struct workqueue_struct * *wq*, struct delayed_work * *dwork*, unsigned long *delay*)
modify delay of or queue a delayed work

Parameters

struct workqueue_struct * wq workqueue to use

struct delayed_work * dwork work to queue

unsigned long delay number of jiffies to wait before queueing

Description

mod_delayed_work_on() on local CPU.

bool **schedule_work_on**(int *cpu*, struct work_struct * *work*)
put work task on a specific cpu

Parameters

int cpu cpu to put the work task on

struct work_struct * work job to be done

Description

This puts a job on a specific cpu

bool **schedule_work**(struct work_struct * *work*)
put work task in global workqueue

Parameters

struct work_struct * 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.

void **flush_scheduled_work**(void)
ensure that any scheduled work has run to completion.

Parameters

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.

bool **schedule_delayed_work_on**(int *cpu*, struct delayed_work * *dwork*, unsigned long *delay*)
queue work in global workqueue on CPU after delay

Parameters

int cpu cpu to use

struct delayed_work * dwork job to be done

unsigned long 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.

bool **`schedule_delayed_work`**(struct delayed_work * *dwork*, unsigned long *delay*)
put work task in global workqueue after delay

Parameters

`struct delayed_work * dwork` job to be done

`unsigned long 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.

bool **`queue_work_on`**(int *cpu*, struct workqueue_struct * *wq*, struct work_struct * *work*)
queue work on specific cpu

Parameters

`int cpu` CPU number to execute work on

`struct workqueue_struct * wq` workqueue to use

`struct work_struct * 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.

bool **`queue_delayed_work_on`**(int *cpu*, struct workqueue_struct * *wq*, struct delayed_work * *dwork*, unsigned long *delay*)
queue work on specific CPU after delay

Parameters

`int cpu` CPU number to execute work on

`struct workqueue_struct * wq` workqueue to use

`struct delayed_work * dwork` work to queue

`unsigned long 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.

bool **`mod_delayed_work_on`**(int *cpu*, struct workqueue_struct * *wq*, struct delayed_work * *dwork*, unsigned long *delay*)
modify delay of or queue a delayed work on specific CPU

Parameters

`int cpu` CPU number to execute work on

`struct workqueue_struct * wq` workqueue to use

`struct delayed_work * dwork` work to queue

`unsigned long 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.

void **`flush_workqueue`**(struct workqueue_struct * *wq*)
ensure that any scheduled work has run to completion.

Parameters

`struct workqueue_struct * 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.

void **`drain_workqueue`**(struct workqueue_struct * *wq*)
drain a workqueue

Parameters

`struct workqueue_struct * 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.

bool **`flush_work`**(struct work_struct * *work*)
wait for a work to finish executing the last queueing instance

Parameters

`struct work_struct * 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.

bool **`cancel_work_sync`**(struct work_struct * *work*)
cancel a work and wait for it to finish

Parameters

`struct work_struct * 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.

bool **`flush_delayed_work`**(struct delayed_work * *dwork*)
wait for a dwork to finish executing the last queueing

Parameters

struct delayed_work * 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.

bool **cancel_delayed_work**(struct delayed_work * *dwork*)
cancel a delayed work

Parameters

struct delayed_work * 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.

bool **cancel_delayed_work_sync**(struct delayed_work * *dwork*)
cancel a delayed work and wait for it to finish

Parameters

struct delayed_work * dwork the delayed work cancel

Description

This is *cancel_work_sync()* for delayed works.

Return

`true` if **dwork** was pending, `false` otherwise.

int **execute_in_process_context**(work_func_t *fn*, struct execute_work * *ew*)
reliably execute the routine with user context

Parameters

work_func_t fn the function to execute

struct execute_work * 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

int **apply_workqueue_attrs**(struct workqueue_struct * *wq*, const struct *workqueue_attrs* * *attrs*)
apply new workqueue_attrs to an unbound workqueue

Parameters

struct workqueue_struct * wq the target workqueue

`const struct workqueue_attrs * attrs` the workqueue_attrs to apply, allocated with `alloc_workqueue_attrs()`

Description

Apply **attrs** to an unbound workqueue **wq**. Unless disabled, on NUMA machines, this function maps a separate pwq to each NUMA node with possibles CPUs in **attrs**->cpumask so that work items are affine to the NUMA node it was issued on. Older pwqs are released as in-flight work items finish. Note that a work item which repeatedly requeues itself back-to-back will stay on its current pwq.

Performs GFP_KERNEL allocations.

Return

0 on success and -errno on failure.

void **`destroy_workqueue`**(struct workqueue_struct * *wq*)
safely terminate a workqueue

Parameters

`struct workqueue_struct * wq` target workqueue

Description

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

void **`workqueue_set_max_active`**(struct workqueue_struct * *wq*, int *max_active*)
adjust max_active of a workqueue

Parameters

`struct workqueue_struct * wq` target workqueue

`int max_active` new max_active value.

Description

Set max_active of **wq** to **max_active**.

Context

Don't call from IRQ context.

struct work_struct * **`current_work`**(void)
retrieve `current` task's work struct

Parameters

`void` no arguments

Description

Determine if `current` task is a workqueue worker and what it's working on. Useful to find out the context that the `current` task is running in.

Return

work struct if `current` task is a workqueue worker, `NULL` otherwise.

bool **`workqueue_congested`**(int *cpu*, struct workqueue_struct * *wq*)
test whether a workqueue is congested

Parameters

`int cpu` CPU in question

`struct workqueue_struct * 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 contested on other CPUs / NUMA nodes.

Return

`true` if congested, `false` otherwise.

unsigned int **`work_busy`**(struct work_struct * *work*)
test whether a work is currently pending or running

Parameters

`struct work_struct * 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.

long **`work_on_cpu`**(int *cpu*, long (*fn) (void *, void * *arg*)
run a function in thread context on a particular cpu

Parameters

`int cpu` the cpu to run on

`long (*)(void *) fn` the function to run

`void * 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.

long **`work_on_cpu_safe`**(int *cpu*, long (*fn) (void *, void * *arg*)
run a function in thread context on a particular cpu

Parameters

`int cpu` the cpu to run on

`long (*)(void *) fn` the function to run

`void * arg` the function argument

Description

Disables CPU hotplug and calls *`work_on_cpu()`*. The caller must not hold any locks which would prevent **fn** from completing.

Return

The value **fn** returns.

Internal Functions

int **`wait_task_stopped`**(struct wait_opts * *wo*, int *ptrace*, struct task_struct * *p*)
Wait for `TASK_STOPPED` or `TASK_TRACED`

Parameters

`struct wait_opts * wo` wait options

`int ptrace` is the wait for ptrace

`struct task_struct * 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.

Return

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.

bool **`task_set_jobctl_pending`**(struct task_struct * *task*, unsigned long *mask*)
set jobctl pending bits

Parameters

`struct task_struct * task` target task

`unsigned long 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.

Return

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

void **`task_clear_jobctl_trapping`**(struct task_struct * *task*)
clear jobctl trapping bit

Parameters

`struct task_struct * 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**->sighand->siglock held.

void **`task_clear_jobctl_pending`**(struct task_struct * *task*, unsigned long *mask*)
clear jobctl pending bits

Parameters

`struct task_struct * task` target task

`unsigned long 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.

bool **task_participate_group_stop**(struct task_struct * *task*)
participate in a group stop

Parameters

struct task_struct * 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**->sighand->siglock held.

Return

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

void **ptrace_trap_notify**(struct task_struct * *t*)
schedule trap to notify ptracer

Parameters

struct task_struct * 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**->sighand->siglock held.

void **do_notify_parent_cldstop**(struct task_struct * *tsk*, bool *for_ptracer*, int *why*)
notify parent of stopped/continued state change

Parameters

struct task_struct * tsk task reporting the state change

bool for_ptracer the notification is for ptracer

int why 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.

bool **do_signal_stop**(int *signr*)
handle group stop for SIGSTOP and other stop signals

Parameters

int 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**->sighand->siglock held, which is released on true return.

Return

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

void **do_jobctl_trap**(void)
take care of ptrace jobctl traps

Parameters

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**->sighand->siglock held, which may be released and re-acquired before returning with intervening sleep.

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

Parameters

struct ksignal * ksig kernel signal struct

int 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.

long **sys_restart_syscall**(void)
restart a system call

Parameters

void no arguments

void **set_current_blocked**(sigset_t * *newset*)
change current->blocked mask

Parameters

sigset_t * 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.

long **sys_rt_sigprocmask**(int *how*, sigset_t __user * *nset*, sigset_t __user * *oset*, size_t *sigsetsize*)
change the list of currently blocked signals

Parameters

int how whether to add, remove, or set signals

sigset_t __user * nset stores pending signals

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

size_t sigsetsize size of sigset_t type

long **sys_rt_sigpending**(sigset_t __user * *uset*, size_t *sigsetsize*)
examine a pending signal that has been raised while blocked

Parameters

sigset_t __user * uset stores pending signals

size_t sigsetsize size of sigset_t type or larger

int **do_sigtimedwait**(const sigset_t * *which*, siginfo_t * *info*, const struct timespec * *ts*)
wait for queued signals specified in **which**

Parameters

const sigset_t * which queued signals to wait for

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

const struct timespec * ts upper bound on process time suspension

long **sys_rt_sigtimedwait**(const sigset_t __user * *uthese*, siginfo_t __user * *uinfo*, const struct timespec __user * *uts*, size_t *sigsetsize*)
synchronously wait for queued signals specified in **uthese**

Parameters

const sigset_t __user * uthese queued signals to wait for

siginfo_t __user * uinfo if non-null, the signal's siginfo is returned here

const struct timespec __user * uts upper bound on process time suspension

size_t sigsetsize size of sigset_t type

long **sys_kill**(pid_t *pid*, int *sig*)
send a signal to a process

Parameters

pid_t pid the PID of the process

int sig signal to be sent

long **sys_tgkill**(pid_t *tgid*, pid_t *pid*, int *sig*)
send signal to one specific thread

Parameters

pid_t tgid the thread group ID of the thread

pid_t pid the PID of the thread

int 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.

long **sys_tkill**(pid_t *pid*, int *sig*)
send signal to one specific task

Parameters

pid_t pid the PID of the task

int sig signal to be sent

Description

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

long **sys_rt_sigqueueinfo**(pid_t *pid*, int *sig*, siginfo_t __user * *uinfo*)
send signal information to a signal

Parameters

pid_t pid the PID of the thread

int sig signal to be sent

siginfo_t __user * uinfo signal info to be sent

long **sys_sigpending**(old_sigset_t __user * *set*)
examine pending signals

Parameters

old_sigset_t __user * set where mask of pending signal is returned

long **sys_sigprocmask**(int *how*, old_sigset_t __user * *nset*, old_sigset_t __user * *oset*)
examine and change blocked signals

Parameters

int how whether to add, remove, or set signals

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

old_sigset_t __user * 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.

long **sys_rt_sigaction**(int *sig*, const struct sigaction __user * *act*, struct sigaction __user * *oact*, size_t *sigsetsize*)
alter an action taken by a process

Parameters

int sig signal to be sent

const struct sigaction __user * act new sigaction

struct sigaction __user * oact used to save the previous sigaction

size_t sigsetsize size of sigset_t type

long **sys_rt_sigsuspend**(sigset_t __user * *unewset*, size_t *sigsetsize*)
replace the signal mask for a value with the **unewset** value until a signal is received

Parameters

sigset_t __user * unewset new signal mask value

size_t sigsetsize size of sigset_t type

kthread_create(*threadfn*, *data*, *namefmt*, *arg...*)
create a kthread on the current node

Parameters

threadfn the function to run in the thread

data data pointer for **threadfn()**

namefmt printf-style format string for the thread name

arg... arguments for **namefmt**.

Description

This macro will create a kthread on the current node, leaving it in the stopped state. This is just a helper for *kthread_create_on_node()*; see the documentation there for more details.

kthread_run(*threadfn*, *data*, *namefmt*, *...*)
create and wake a thread.

Parameters

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).

bool **kthread_should_stop**(void)
should this kthread return now?

Parameters

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()*.

bool **kthread_should_park**(void)
should this kthread park now?

Parameters

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.

bool **kthread_freezable_should_stop**(bool * *was_frozen*)
should this freezable kthread return now?

Parameters

bool * 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.

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

create a kthread.

Parameters

int (*)(void *data) threadfn the function to run until signal_pending(current).

void * data data ptr for **threadfn**.

int node task and thread structures for the thread are allocated on this node

const char namefmt 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).

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

bind a just-created kthread to a cpu.

Parameters

struct task_struct * p thread created by *kthread_create()*.

unsigned int 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()*).

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

unpark a thread created by *kthread_create()*.

Parameters

struct task_struct * 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.

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

park a thread created by *kthread_create()*.

Parameters

struct task_struct * 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.

int **kthread_stop**(struct task_struct * *k*)
stop a thread created by *kthread_create()*.

Parameters

struct task_struct * 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.

int **kthread_worker_fn**(void * *worker_ptr*)
kthread function to process kthread_worker

Parameters

void * worker_ptr pointer to initialized kthread_worker

Description

This function implements the main cycle of kthread worker. It processes work_list until it is stopped with *kthread_stop()*. It sleeps when the queue is empty.

The works are not allowed to keep any locks, disable preemption or interrupts when they finish. There is defined a safe point for freezing when one work finishes and before a new one is started.

Also the works must not be handled by more than one worker at the same time, see also *kthread_queue_work()*.

struct kthread_worker * **kthread_create_worker**(unsigned int *flags*, const char *namefmt*, ...)
create a kthread worker

Parameters

unsigned int flags flags modifying the default behavior of the worker

const char namefmt printf-style name for the kthread worker (task).

... variable arguments

Description

Returns a pointer to the allocated worker on success, ERR_PTR(-ENOMEM) when the needed structures could not get allocated, and ERR_PTR(-EINTR) when the worker was SIGKILLed.

struct kthread_worker * **kthread_create_worker_on_cpu**(int *cpu*, unsigned int *flags*, const char *namefmt*, ...)
create a kthread worker and bind it it to a given CPU and the associated NUMA node.

Parameters

int cpu CPU number

unsigned int flags flags modifying the default behavior of the worker

const char namefmt printf-style name for the kthread worker (task).

... variable arguments

Description

Use a valid CPU number if you want to bind the kthread worker to the given CPU and the associated NUMA node.

A good practice is to add the cpu number also into the worker name. For example, use kthread_create_worker_on_cpu(cpu, "helper/d", cpu).

Returns a pointer to the allocated worker on success, ERR_PTR(-ENOMEM) when the needed structures could not get allocated, and ERR_PTR(-EINTR) when the worker was SIGKILLed.

bool **`kthread_queue_work`**(struct kthread_worker * *worker*, struct kthread_work * *work*)
queue a kthread_work

Parameters

`struct kthread_worker * worker` target kthread_worker

`struct kthread_work * 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.

Reinitialize the work if it needs to be used by another worker. For example, when the worker was stopped and started again.

void **`kthread_delayed_work_timer_fn`**(struct timer_list * *t*)
callback that queues the associated kthread delayed work when the timer expires.

Parameters

`struct timer_list * t` pointer to the expired timer

Description

The format of the function is defined by struct timer_list. It should have been called from irqsafe timer with irq already off.

bool **`kthread_queue_delayed_work`**(struct kthread_worker * *worker*, struct kthread_delayed_work * *dwork*, unsigned long *delay*)
queue the associated kthread work after a delay.

Parameters

`struct kthread_worker * worker` target kthread_worker

`struct kthread_delayed_work * dwork` kthread_delayed_work to queue

`unsigned long delay` number of jiffies to wait before queuing

Description

If the work has not been pending it starts a timer that will queue the work after the given **delay**. If **delay** is zero, it queues the work immediately.

Return

`false` if the **work** has already been pending. It means that either the timer was running or the work was queued. It returns `true` otherwise.

void **`kthread_flush_work`**(struct kthread_work * *work*)
flush a kthread_work

Parameters

`struct kthread_work * work` work to flush

Description

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

bool **kthread_mod_delayed_work**(struct kthread_worker * *worker*, struct kthread_delayed_work * *dwork*, unsigned long *delay*)
modify delay of or queue a kthread delayed work

Parameters

struct kthread_worker * worker kthread worker to use

struct kthread_delayed_work * dwork kthread delayed work to queue

unsigned long delay number of jiffies to wait before queuing

Description

If **dwork** is idle, equivalent to *kthread_queue_delayed_work()*. Otherwise, modify **dwork**'s timer so that it expires after **delay**. If **delay** is zero, **work** is guaranteed to be queued immediately.

Return

`true` if **dwork** was pending and its timer was modified, `false` otherwise.

A special case is when the work is being canceled in parallel. It might be caused either by the real *kthread_cancel_delayed_work_sync()* or yet another *kthread_mod_delayed_work()* call. We let the other command win and return `false` here. The caller is supposed to synchronize these operations a reasonable way.

This function is safe to call from any context including IRQ handler. See `__kthread_cancel_work()` and *kthread_delayed_work_timer_fn()* for details.

bool **kthread_cancel_work_sync**(struct kthread_work * *work*)
cancel a kthread work and wait for it to finish

Parameters

struct kthread_work * work the kthread 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. On return from this function, **work** is guaranteed to be not pending or executing on any CPU.

kthread_cancel_work_sync(`delayed_work->work`) must not be used for delayed_work's. Use *kthread_cancel_delayed_work_sync()* instead.

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

Return

`true` if **work** was pending, `false` otherwise.

bool **kthread_cancel_delayed_work_sync**(struct kthread_delayed_work * *dwork*)
cancel a kthread delayed work and wait for it to finish.

Parameters

struct kthread_delayed_work * dwork the kthread delayed work to cancel

Description

This is *kthread_cancel_work_sync()* for delayed works.

Return

`true` if **dwork** was pending, `false` otherwise.

void **kthread_flush_worker**(struct kthread_worker * *worker*)
flush all current works on a kthread_worker

Parameters

struct kthread_worker * worker worker to flush

Description

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

void **kthread_destroy_worker**(struct kthread_worker * *worker*)
destroy a kthread worker

Parameters

struct kthread_worker * worker worker to be destroyed

Description

Flush and destroy **worker**. The simple flush is enough because the kthread worker API is used only in trivial scenarios. There are no multi-step state machines needed.

void **kthread_associate_blkcg**(struct cgroup_subsys_state * *css*)
associate blkcg to current kthread

Parameters

struct cgroup_subsys_state * css the cgroup info

Description

Current thread must be a kthread. The thread is running jobs on behalf of other threads. In some cases, we expect the jobs attach cgroup info of original threads instead of that of current thread. This function stores original thread's cgroup info in current kthread context for later retrieval.

struct cgroup_subsys_state * **kthread_blkcg**(void)
get associated blkcg css of current kthread

Parameters

void no arguments

Description

Current thread must be a kthread.

Reference counting

struct **refcount_struct**
variant of atomic_t specialized for reference counts

Definition

```
struct refcount_struct {
  atomic_t refs;
};
```

Members

refs atomic_t counter field

Description

The counter saturates at UINT_MAX and will not move once there. This avoids wrapping the counter and causing 'spurious' use-after-free bugs.

void **refcount_set**(refcount_t * *r*, unsigned int *n*)
set a refcount's value

Parameters

refcount_t * r the refcount

unsigned int n value to which the refcount will be set

unsigned int **`refcount_read`**(const refcount_t * *r*)
    get a refcount's value

Parameters

`const refcount_t * r` the refcount

Return

the refcount's value

bool **`refcount_add_not_zero`**(unsigned int *i*, refcount_t * *r*)
    add a value to a refcount unless it is 0

Parameters

`unsigned int i` the value to add to the refcount

`refcount_t * r` the refcount

Description

Will saturate at UINT_MAX and WARN.

Provides no memory ordering, it is assumed the caller has guaranteed the object memory to be stable (RCU, etc.). It does provide a control dependency and thereby orders future stores. See the comment on top.

Use of this function is not recommended for the normal reference counting use case in which references are taken and released one at a time. In these cases, *`refcount_inc()`*, or one of its variants, should instead be used to increment a reference count.

Return

false if the passed refcount is 0, true otherwise

void **`refcount_add`**(unsigned int *i*, refcount_t * *r*)
    add a value to a refcount

Parameters

`unsigned int i` the value to add to the refcount

`refcount_t * r` the refcount

Description

Similar to *`atomic_add()`*, but will saturate at UINT_MAX and WARN.

Provides no memory ordering, it is assumed the caller has guaranteed the object memory to be stable (RCU, etc.). It does provide a control dependency and thereby orders future stores. See the comment on top.

Use of this function is not recommended for the normal reference counting use case in which references are taken and released one at a time. In these cases, *`refcount_inc()`*, or one of its variants, should instead be used to increment a reference count.

bool **`refcount_inc_not_zero`**(refcount_t * *r*)
    increment a refcount unless it is 0

Parameters

`refcount_t * r` the refcount to increment

Description

Similar to `atomic_inc_not_zero()`, but will saturate at UINT_MAX and WARN.

Provides no memory ordering, it is assumed the caller has guaranteed the object memory to be stable (RCU, etc.). It does provide a control dependency and thereby orders future stores. See the comment on top.

Return

true if the increment was successful, false otherwise

void **`refcount_inc`**(refcount_t * *r*)
    increment a refcount

Parameters

`refcount_t * r` the refcount to increment

Description

Similar to *`atomic_inc()`*, but will saturate at UINT_MAX and WARN.

Provides no memory ordering, it is assumed the caller already has a reference on the object.

Will WARN if the refcount is 0, as this represents a possible use-after-free condition.

bool **`refcount_sub_and_test`**(unsigned int *i*, refcount_t * *r*)
    subtract from a refcount and test if it is 0

Parameters

`unsigned int i` amount to subtract from the refcount

`refcount_t * r` the refcount

Description

Similar to *`atomic_dec_and_test()`*, but it will WARN, return false and ultimately leak on underflow and will fail to decrement when saturated at UINT_MAX.

Provides release memory ordering, such that prior loads and stores are done before, and provides a control dependency such that `free()` must come after. See the comment on top.

Use of this function is not recommended for the normal reference counting use case in which references are taken and released one at a time. In these cases, *`refcount_dec()`*, or one of its variants, should instead be used to decrement a reference count.

Return

true if the resulting refcount is 0, false otherwise

bool **`refcount_dec_and_test`**(refcount_t * *r*)
    decrement a refcount and test if it is 0

Parameters

`refcount_t * r` the refcount

Description

Similar to *`atomic_dec_and_test()`*, it will WARN on underflow and fail to decrement when saturated at UINT_MAX.

Provides release memory ordering, such that prior loads and stores are done before, and provides a control dependency such that `free()` must come after. See the comment on top.

Return

true if the resulting refcount is 0, false otherwise

void **`refcount_dec`**(refcount_t * *r*)
    decrement a refcount

Parameters

`refcount_t * r` the refcount

Description

Similar to *`atomic_dec()`*, it will WARN on underflow and fail to decrement when saturated at UINT_MAX.

Provides release memory ordering, such that prior loads and stores are done before.

bool **refcount_dec_if_one**(refcount_t * *r*)
decrement a refcount if it is 1

Parameters

refcount_t * r the refcount

Description

No atomic_t counterpart, it attempts a 1 -> 0 transition and returns the success thereof.

Like all decrement operations, it provides release memory order and provides a control dependency.

It can be used like a try-delete operator; this explicit case is provided and not cmpxchg in generic, because that would allow implementing unsafe operations.

Return

true if the resulting refcount is 0, false otherwise

bool **refcount_dec_not_one**(refcount_t * *r*)
decrement a refcount if it is not 1

Parameters

refcount_t * r the refcount

Description

No atomic_t counterpart, it decrements unless the value is 1, in which case it will return false.

Was often done like: atomic_add_unless(`var`, -1, 1)

Return

true if the decrement operation was successful, false otherwise

bool **refcount_dec_and_mutex_lock**(refcount_t * *r*, struct mutex * *lock*)
return holding mutex if able to decrement refcount to 0

Parameters

refcount_t * r the refcount

struct mutex * lock the mutex to be locked

Description

Similar to `atomic_dec_and_mutex_lock()`, it will WARN on underflow and fail to decrement when saturated at UINT_MAX.

Provides release memory ordering, such that prior loads and stores are done before, and provides a control dependency such that `free()` must come after. See the comment on top.

Return

true and hold mutex if able to decrement refcount to 0, false otherwise

bool **refcount_dec_and_lock**(refcount_t * *r*, spinlock_t * *lock*)
return holding spinlock if able to decrement refcount to 0

Parameters

refcount_t * r the refcount

spinlock_t * lock the spinlock to be locked

Description

Similar to `atomic_dec_and_lock()`, it will WARN on underflow and fail to decrement when saturated at UINT_MAX.

Provides release memory ordering, such that prior loads and stores are done before, and provides a control dependency such that `free()` must come after. See the comment on top.

Return

true and hold spinlock if able to decrement refcount to 0, false otherwise

Atomics

int **atomic_read**(const atomic_t * *v*)
read atomic variable

Parameters

const atomic_t * v pointer of type atomic_t

Description

Atomically reads the value of **v**.

void **atomic_set**(atomic_t * *v*, int *i*)
set atomic variable

Parameters

atomic_t * v pointer of type atomic_t

int i required value

Description

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

void **atomic_add**(int *i*, atomic_t * *v*)
add integer to atomic variable

Parameters

int i integer value to add

atomic_t * v pointer of type atomic_t

Description

Atomically adds **i** to **v**.

void **atomic_sub**(int *i*, atomic_t * *v*)
subtract integer from atomic variable

Parameters

int i integer value to subtract

atomic_t * v pointer of type atomic_t

Description

Atomically subtracts **i** from **v**.

bool **atomic_sub_and_test**(int *i*, atomic_t * *v*)
subtract value from variable and test result

Parameters

int i integer value to subtract

atomic_t * 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.

void **atomic_inc**(atomic_t * *v*)
increment atomic variable

Parameters

`atomic_t * v` pointer of type atomic_t

Description

Atomically increments **v** by 1.

void **`atomic_dec`**(atomic_t * *v*)
    decrement atomic variable

Parameters

`atomic_t * v` pointer of type atomic_t

Description

Atomically decrements **v** by 1.

bool **`atomic_dec_and_test`**(atomic_t * *v*)
    decrement and test

Parameters

`atomic_t * 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.

bool **`atomic_inc_and_test`**(atomic_t * *v*)
    increment and test

Parameters

`atomic_t * 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.

bool **`atomic_add_negative`**(int *i*, atomic_t * *v*)
    add and test if negative

Parameters

`int i` integer value to add

`atomic_t * 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.

int **`atomic_add_return`**(int *i*, atomic_t * *v*)
    add integer and return

Parameters

`int i` integer value to add

`atomic_t * v` pointer of type atomic_t

Description

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

int **`atomic_sub_return`**(int *i*, atomic_t * *v*)
    subtract integer and return

Parameters

`int i` integer value to subtract

atomic_t * v pointer of type atomic_t

Description

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

int **__atomic_add_unless**(atomic_t * *v*, int *a*, int *u*)
add unless the number is already a given value

Parameters

int a the amount to add to v...

atomic_t * v pointer of type atomic_t

int a the amount to add to v...

int 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**.

Kernel objects manipulation

char * **kobject_get_path**(struct kobject * *kobj*, gfp_t *gfp_mask*)
generate and return the path associated with a given kobj and kset pair.

Parameters

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

gfp_t gfp_mask the allocation type used to allocate the path

Description

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

int **kobject_set_name**(struct kobject * *kobj*, const char * *fmt*, ...)
Set the name of a kobject

Parameters

struct kobject * kobj struct kobject to set the name of

const char * 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.

void **kobject_init**(struct kobject * *kobj*, struct kobj_type * *ktype*)
initialize a kobject structure

Parameters

struct kobject * kobj pointer to the kobject to initialize

struct kobj_type * 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.

int **kobject_add**(struct kobject * *kobj*, struct kobject * *parent*, const char * *fmt*, ...)
the main kobject add function

Parameters

struct kobject * kobj the kobject to add

struct kobject * parent pointer to the parent of the kobject.

const char * fmt 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.

int **kobject_init_and_add**(struct kobject * *kobj*, struct kobj_type * *ktype*, struct kobject * *parent*, const char * *fmt*, ...)
initialize a kobject structure and add it to the kobject hierarchy

Parameters

struct kobject * kobj pointer to the kobject to initialize

struct kobj_type * ktype pointer to the ktype for this kobject.

struct kobject * parent pointer to the parent of this kobject.

const char * fmt 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.

int **kobject_rename**(struct kobject * *kobj*, const char * *new_name*)
change the name of an object

Parameters

struct kobject * kobj object in question.

const char * 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.

int **kobject_move**(struct kobject * *kobj*, struct kobject * *new_parent*)
move object to another parent

Parameters

struct kobject * kobj object in question.

struct kobject * new_parent object's new parent (can be NULL)

void **kobject_del**(struct kobject * *kobj*)
unlink kobject from hierarchy.

Parameters

struct kobject * kobj object.

struct kobject * **kobject_get**(struct kobject * *kobj*)
increment refcount for object.

Parameters

struct kobject * kobj object.

void **kobject_put**(struct kobject * *kobj*)
decrement refcount for object.

Parameters

struct kobject * kobj object.

Description

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

struct kobject * **kobject_create_and_add**(const char * *name*, struct kobject * *parent*)
create a struct kobject dynamically and register it with sysfs

Parameters

const char * name the name for the kobject

struct 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.

int **kset_register**(struct kset * *k*)
initialize and add a kset.

Parameters

struct kset * k kset.

void **kset_unregister**(struct kset * *k*)
remove a kset.

Parameters

struct kset * k kset.

struct kobject * **kset_find_obj**(struct kset * *kset*, const char * *name*)
search for object in kset.

Parameters

struct kset * kset kset we're looking in.

const char * name object's name.

Description

Lock kset via **kset**->subsys, and iterate over **kset**->list, looking for a matching kobject. If matching object is found take a reference and return the object.

struct kset * **kset_create_and_add**(const char * *name*, const struct kset_uevent_ops * *uevent_ops*, struct kobject * *parent_kobj*)
create a struct kset dynamically and add it to sysfs

Parameters

const char * name the name for the kset

const struct kset_uevent_ops * uevent_ops a struct kset_uevent_ops for the kset

struct kobject * 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

REPEAT_BYTE(*x*)
    repeat the value **x** multiple times as an unsigned long value

Parameters

x value to repeat

NOTE

x is not checked for > 0xff; larger values produce odd results.

ARRAY_SIZE(*arr*)
    get the number of elements in array **arr**

Parameters

arr array to be sized

FIELD_SIZEOF(*t*, *f*)
    get the size of a struct's field

Parameters

t the target struct

f the target struct's field

Return

the size of **f** in the struct definition without having a declared instance of **t**.

upper_32_bits(*n*)
    return bits 32-63 of a number

Parameters

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 >= width of type" warning when that quantity is 32-bits.

lower_32_bits(*n*)
    return bits 0-31 of a number

Parameters

n the number we're accessing

might_sleep()
    annotation for functions that can sleep

Parameters

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.

abs(*x*)
return absolute value of an argument

Parameters

x the value. If it is unsigned type, it is converted to signed type first. char is treated as if it was signed (regardless of whether it really is) but the macro's return type is preserved as char.

Return

an absolute value of x.

u32 **reciprocal_scale**(u32 *val*, u32 *ep_ro*)
"scale" a value into range [0, ep_ro)

Parameters

u32 val value

u32 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**).

int **kstrtoul**(const char * *s*, unsigned int *base*, unsigned long * *res*)
convert a string to an unsigned long

Parameters

const char * 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.

unsigned int 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.

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

Description

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

int **kstrtol**(const char * *s*, unsigned int *base*, long * *res*)
convert a string to a long

Parameters

const char * s The start of the string. The string must be null-terminated, and may also include a single newline before its terminating null. The first character may also be a plus sign or a minus sign.

unsigned int 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.

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

Description

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

trace_printk(*fmt*, *...*)
printf formatting in the ftrace buffer

Parameters

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 optization 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.

trace_puts(*str*)
write a string into the ftrace buffer

Parameters

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" effects, 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.)

Return

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

min(*x*, *y*)
return minimum of two values of the same or compatible types

Parameters

x first value

y second value

max(*x*, *y*)
return maximum of two values of the same or compatible types

Parameters

x first value

y second value

min3(*x*, *y*, *z*)
return minimum of three values

Parameters

x first value

y second value

z third value

max3(*x*, *y*, *z*)
return maximum of three values

Parameters

x first value

y second value

z third value

min_not_zero(*x*, *y*)
return the minimum that is _not_ zero, unless both are zero

Parameters

x value1

y value2

clamp(*val*, *lo*, *hi*)
return a value clamped to a given range with strict typechecking

Parameters

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.

min_t(*type*, *x*, *y*)
return minimum of two values, using the specified type

Parameters

type data type to use

x first value

y second value

max_t(*type*, *x*, *y*)
return maximum of two values, using the specified type

Parameters

type data type to use

x first value

y second value

clamp_t(*type*, *val*, *lo*, *hi*)
return a value clamped to a given range using a given type

Parameters

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.

clamp_val(*val*, *lo*, *hi*)
return a value clamped to a given range using val's type

Parameters

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 **lo** and **hi** are literals that will otherwise be assigned a signed integer type.

swap(*a*, *b*)
swap values of **a** and **b**

Parameters

a first value

b second value

container_of(*ptr*, *type*, *member*)
cast a member of a structure out to the containing structure

Parameters

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.

__visible int **printk**(const char * *fmt*, ...)
print a kernel message

Parameters

const char * 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.

void **console_lock**(void)
    lock the console system for exclusive use.

Parameters

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.

int **console_trylock**(void)
    try to lock the console system for exclusive use.

Parameters

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.

void **console_unlock**(void)
    unlock the console system

Parameters

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.

void __sched **console_conditional_schedule**(void)
    yield the CPU if required

Parameters

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()*;.

bool **printk_timed_ratelimit**(unsigned long * *caller_jiffies*, unsigned int *interval_msecs*)
caller-controlled printk ratelimiting

Parameters

unsigned long * caller_jiffies pointer to caller's state

unsigned int 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.

int **kmsg_dump_register**(struct kmsg_dumper * *dumper*)
register a kernel log dumper.

Parameters

struct kmsg_dumper * 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.

int **kmsg_dump_unregister**(struct kmsg_dumper * *dumper*)
unregister a kmsg dumper.

Parameters

struct kmsg_dumper * dumper pointer to the kmsg_dumper structure

Description

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

bool **kmsg_dump_get_line**(struct kmsg_dumper * *dumper*, bool *syslog*, char * *line*, size_t *size*, size_t * *len*)
retrieve one kmsg log line

Parameters

struct kmsg_dumper * dumper registered kmsg dumper

bool syslog include the "<4>" prefixes

char * line buffer to copy the line to

size_t size maximum size of the buffer

size_t * 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.

bool **kmsg_dump_get_buffer**(struct kmsg_dumper * *dumper*, bool *syslog*, char * *buf*, size_t *size*, size_t * *len*)
copy kmsg log lines

Parameters

struct kmsg_dumper * dumper registered kmsg dumper

bool syslog include the "<4>" prefixes

char * buf buffer to copy the line to

size_t size maximum size of the buffer

size_t * 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.

void **kmsg_dump_rewind**(struct kmsg_dumper * *dumper*)
    reset the interator

Parameters

struct kmsg_dumper * 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.:c:func:*dump()* callback.

void **panic**(const char * *fmt*, ...)
    halt the system

Parameters

const char * fmt The text string to print

... variable arguments

Description

    Display a message, then perform cleanups.

    This function never returns.

void **add_taint**(unsigned *flag*, enum lockdep_ok *lockdep_ok*)

Parameters

unsigned flag one of the TAINT_* constants.

enum lockdep_ok lockdep_ok whether lock debugging is still OK.

Description

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

bool notrace **rcu_is_watching**(void)
    see if RCU thinks that the current CPU is idle

Parameters

void no arguments

Description

Return true if RCU is watching the running CPU, which means that this CPU can safely enter RCU read-side critical sections. In other words, if the current CPU is in its idle loop and is neither in an interrupt or NMI handler, return true.

void **call_rcu_sched**(struct rcu_head * *head*, rcu_callback_t *func*)
    Queue an RCU for invocation after sched grace period.

Parameters

`struct rcu_head * head` structure to be used for queueing the RCU updates.

`rcu_callback_t func` actual callback function to be invoked after the grace period

Description

The callback function will be invoked some time after a full grace period elapses, in other words after all currently executing RCU read-side critical sections have completed. *`call_rcu_sched()`* assumes that the read-side critical sections end on enabling of preemption or on voluntary preemption. RCU read-side critical sections are delimited by:

- `rcu_read_lock_sched()` and `rcu_read_unlock_sched()`, OR
- anything that disables preemption.

  These may be nested.

See the description of *`call_rcu()`* for more detailed information on memory ordering guarantees.

void **`call_rcu_bh`**(struct rcu_head * *head*, rcu_callback_t *func*)
    Queue an RCU for invocation after a quicker grace period.

Parameters

`struct rcu_head * head` structure to be used for queueing the RCU updates.

`rcu_callback_t func` actual callback function to be invoked after the grace period

Description

The callback function will be invoked some time after a full grace period elapses, in other words after all currently executing RCU read-side critical sections have completed. *`call_rcu_bh()`* assumes that the read-side critical sections end on completion of a softirq handler. This means that read-side critical sections in process context must not be interrupted by softirqs. This interface is to be used when most of the read-side critical sections are in softirq context. RCU read-side critical sections are delimited by:

- `rcu_read_lock()` and `rcu_read_unlock()`, if in interrupt context, OR
- `rcu_read_lock_bh()` and `rcu_read_unlock_bh()`, if in process context.

These may be nested.

See the description of *`call_rcu()`* for more detailed information on memory ordering guarantees.

void **`synchronize_sched`**(void)
    wait until an rcu-sched grace period has elapsed.

Parameters

`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).

void **synchronize_rcu_bh**(void)
wait until an rcu_bh grace period has elapsed.

Parameters

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.

unsigned long **get_state_synchronize_rcu**(void)
Snapshot current RCU state

Parameters

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.

void **cond_synchronize_rcu**(unsigned long *oldstate*)
Conditionally wait for an RCU grace period

Parameters

unsigned long 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.

unsigned long **get_state_synchronize_sched**(void)
Snapshot current RCU-sched state

Parameters

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.

void **cond_synchronize_sched**(unsigned long *oldstate*)
Conditionally wait for an RCU-sched grace period

Parameters

unsigned long 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.

void **rcu_barrier_bh**(void)
Wait until all in-flight *call_rcu_bh()* callbacks complete.

Parameters

void no arguments

void **rcu_barrier_sched**(void)
Wait for in-flight *call_rcu_sched()* callbacks.

Parameters

void no arguments

void **call_rcu**(struct rcu_head * *head*, rcu_callback_t *func*)
Queue an RCU callback for invocation after a grace period.

Parameters

struct rcu_head * head structure to be used for queueing the RCU updates.

rcu_callback_t func actual callback function to be invoked after the grace period

Description

The callback function will be invoked some time after a full grace period elapses, in other words after all pre-existing RCU read-side critical sections have completed. However, the callback function might well execute concurrently with RCU read-side critical sections that started after *call_rcu()* was invoked. RCU read-side critical sections are delimited by `rcu_read_lock()` and `rcu_read_unlock()`, and may be nested.

Note that all CPUs must agree that the grace period extended beyond all pre-existing RCU read-side critical section. On systems with more than one CPU, this means that when "`func()`" is invoked, each CPU is guaranteed to have executed a full memory barrier since the end of its last RCU read-side critical section whose beginning preceded the call to *call_rcu()*. It also means that each CPU executing an RCU read-side critical section that continues beyond the start of "`func()`" must have executed a memory barrier after the *call_rcu()* but 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 *call_rcu()* and CPU B invoked the resulting RCU callback function "`func()`", then both CPU A and CPU B are guaranteed to execute a full memory barrier during the time interval between the call to *call_rcu()* and the invocation of "`func()`" – even if CPU A and CPU B are the same CPU (but again only if the system has more than one CPU).

void **synchronize_rcu**(void)
wait until a grace period has elapsed.

Parameters

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. However, please note that -only- the memory-ordering guarantees apply. For example, *`synchronize_rcu()`* is -not- guaranteed to wait on things like code protected by `preempt_disable()`, instead, *`synchronize_rcu()`* is -only- guaranteed to wait on RCU read-side critical sections, that is, sections of code protected by `rcu_read_lock()`.

void **`rcu_barrier`**(void)
Wait until all in-flight *`call_rcu()`* callbacks complete.

Parameters

`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.

int **`rcu_read_lock_sched_held`**(void)
might we be in RCU-sched read-side critical section?

Parameters

`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.

void **`rcu_expedite_gp`**(void)
Expedite future RCU grace periods

Parameters

`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.

void **`rcu_unexpedite_gp`**(void)
Cancel prior *`rcu_expedite_gp()`* invocation

Parameters

`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.

int **rcu_read_lock_held**(void)
might we be in RCU read-side critical section?

Parameters

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.

int **rcu_read_lock_bh_held**(void)
might we be in RCU-bh read-side critical section?

Parameters

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.

void **wakeme_after_rcu**(struct rcu_head * *head*)
Callback function to awaken a task after grace period

Parameters

struct rcu_head * head Pointer to rcu_head member within rcu_synchronize structure

Description

Awaken the corresponding task now that a grace period has elapsed.

void **init_rcu_head_on_stack**(struct rcu_head * *head*)
initialize on-stack rcu_head for debugobjects

Parameters

struct rcu_head * 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.

void **destroy_rcu_head_on_stack**(struct rcu_head * *head*)
destroy on-stack rcu_head for debugobjects

Parameters

struct rcu_head * 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.

void **call_rcu_tasks**(struct rcu_head * *rhp*, rcu_callback_t *func*)
Queue an RCU for invocation task-based grace period

Parameters

struct rcu_head * rhp structure to be used for queueing the RCU updates.

rcu_callback_t func actual callback function to be invoked after the grace period

Description

The callback function will be invoked some time after a full grace period elapses, in other words after all currently executing RCU read-side critical sections have completed. *call_rcu_tasks()* assumes that the read-side critical sections end at a voluntary context switch (not a preemption!), entry into idle, or transition to usermode execution. As such, there are no read-side primitives analogous to `rcu_read_lock()` and `rcu_read_unlock()` because this primitive is intended to determine that all tasks have passed through a safe state, not so much for data-strcuture synchronization.

See the description of *call_rcu()* for more detailed information on memory ordering guarantees.

void **synchronize_rcu_tasks**(void)
wait until an rcu-tasks grace period has elapsed.

Parameters

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).

void **rcu_barrier_tasks**(void)
Wait for in-flight *call_rcu_tasks()* callbacks.

Parameters

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

void * **devres_alloc_node**(dr_release_t *release*, size_t *size*, gfp_t *gfp*, int *nid*)
Allocate device resource data

Parameters

dr_release_t release Release function devres will be associated with

size_t size Allocation size

gfp_t gfp Allocation flags

int 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.

Return

Pointer to allocated devres on success, NULL on failure.

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*)
Resource iterator

Parameters

struct device * dev Device to iterate resource from

dr_release_t release Look for resources associated with this release function

dr_match_t match Match function (optional)

void * match_data Data for the match function

void (*)(struct device *, void *, void *) fn Function to be called for each matched resource.

void * data Data for **fn**, the 3rd parameter of **fn**

Description

Call **fn** for each devres of **dev** which is associated with **release** and for which **match** returns 1.

Return

void

void **devres_free**(void * *res*)
Free device resource data

Parameters

void * res Pointer to devres data to free

Description

Free devres created with `devres_alloc()`.

void **devres_add**(struct *device* * *dev*, void * *res*)
Register device resource

Parameters

struct device * dev Device to add resource to

void * 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.

void * **devres_find**(struct *device* * *dev*, dr_release_t *release*, dr_match_t *match*, void * *match_data*)
Find device resource

Parameters

struct device * dev Device to lookup resource from

dr_release_t release Look for resources associated with this release function

dr_match_t match Match function (optional)

void * match_data 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.

Return

Pointer to found devres, NULL if not found.

void * **devres_get**(struct *device* * *dev*, void * *new_res*, dr_match_t *match*, void * *match_data*)
Find devres, if non-existent, add one atomically

Parameters

struct device * dev Device to lookup or add devres for

void * new_res Pointer to new initialized devres to add if not found

dr_match_t match Match function (optional)

void * match_data 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.

Return

Pointer to found or added devres.

void * **devres_remove**(struct *device* * *dev*, dr_release_t *release*, dr_match_t *match*, void * *match_data*)
Find a device resource and remove it

Parameters

struct device * dev Device to find resource from

dr_release_t release Look for resources associated with this release function

dr_match_t match Match function (optional)

void * match_data 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.

Return

Pointer to removed devres on success, NULL if not found.

int **`devres_destroy`**(struct *device* * *dev*, dr_release_t *release*, dr_match_t *match*, void * *match_data*)
Find a device resource and destroy it

Parameters

`struct device * dev` Device to find resource from

`dr_release_t release` Look for resources associated with this release function

`dr_match_t match` Match function (optional)

`void * match_data` 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.

Return

0 if devres is found and freed, -ENOENT if not found.

int **`devres_release`**(struct *device* * *dev*, dr_release_t *release*, dr_match_t *match*, void * *match_data*)
Find a device resource and destroy it, calling release

Parameters

`struct device * dev` Device to find resource from

`dr_release_t release` Look for resources associated with this release function

`dr_match_t match` Match function (optional)

`void * match_data` 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.

Return

0 if devres is found and freed, -ENOENT if not found.

void * **`devres_open_group`**(struct *device* * *dev*, void * *id*, gfp_t *gfp*)
Open a new devres group

Parameters

`struct device * dev` Device to open devres group for

`void * id` Separator ID

`gfp_t 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.

Return

ID of the new group, NULL on failure.

void **devres_close_group**(struct *device* * *dev*, void * *id*)
    Close a devres group

Parameters

struct device * dev Device to close devres group for

void * 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.

void **devres_remove_group**(struct *device* * *dev*, void * *id*)
    Remove a devres group

Parameters

struct device * dev Device to remove group for

void * 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.

int **devres_release_group**(struct *device* * *dev*, void * *id*)
    Release resources in a devres group

Parameters

struct device * dev Device to release group for

void * 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.

Return

The number of released non-group resources.

int **devm_add_action**(struct *device* * *dev*, void (*action) (void *, void * *data*)
    add a custom action to list of managed resources

Parameters

struct device * dev Device that owns the action

void (*)(void *) action Function that should be called

void * 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.

void **devm_remove_action**(struct *device* * *dev*, void (*action) (void *, void * *data*)
    removes previously added custom action

Parameters

struct device * dev Device that owns the action

void (*)(void *) action Function implementing the action

void * 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.

void * **`devm_kmalloc`**(struct *device* * *dev*, size_t *size*, gfp_t *gfp*)
Resource-managed kmalloc

Parameters

`struct device * dev` Device to allocate memory for

`size_t size` Allocation size

`gfp_t 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.

Return

Pointer to allocated memory on success, NULL on failure.

char * **`devm_kstrdup`**(struct *device* * *dev*, const char * *s*, gfp_t *gfp*)
Allocate resource managed space and copy an existing string into that.

Parameters

`struct device * dev` Device to allocate memory for

`const char * s` the string to duplicate

`gfp_t gfp` the GFP mask used in the *`devm_kmalloc()`* call when allocating memory

Return

Pointer to allocated string on success, NULL on failure.

char * **`devm_kvasprintf`**(struct *device* * *dev*, gfp_t *gfp*, const char * *fmt*, va_list *ap*)
Allocate resource managed space and format a string into that.

Parameters

`struct device * dev` Device to allocate memory for

`gfp_t gfp` the GFP mask used in the *`devm_kmalloc()`* call when allocating memory

`const char * fmt` The `printf()`-style format string

`va_list ap` Arguments for the format string

Return

Pointer to allocated string on success, NULL on failure.

char * **`devm_kasprintf`**(struct *device* * *dev*, gfp_t *gfp*, const char * *fmt*, ...)
Allocate resource managed space and format a string into that.

Parameters

`struct device * dev` Device to allocate memory for

`gfp_t gfp` the GFP mask used in the *`devm_kmalloc()`* call when allocating memory

`const char * fmt` The `printf()`-style format string

`...` Arguments for the format string

Return

Pointer to allocated string on success, NULL on failure.

void **devm_kfree**(struct *device* * *dev*, void * *p*)
Resource-managed kfree

Parameters

struct device * dev Device this memory belongs to

void * p Memory to free

Description

Free memory allocated with *devm_kmalloc()*.

void * **devm_kmemdup**(struct *device* * *dev*, const void * *src*, size_t *len*, gfp_t *gfp*)
Resource-managed kmemdup

Parameters

struct device * dev Device this memory belongs to

const void * src Memory region to duplicate

size_t len Memory region length

gfp_t gfp GFP mask to use

Description

Duplicate region of a memory using resource managed kmalloc

unsigned long **devm_get_free_pages**(struct *device* * *dev*, gfp_t *gfp_mask*, unsigned int *order*)
Resource-managed __get_free_pages

Parameters

struct device * dev Device to allocate memory for

gfp_t gfp_mask Allocation gfp flags

unsigned int order Allocation size is (1 << order) pages

Description

Managed get_free_pages. Memory allocated with this function is automatically freed on driver detach.

Return

Address of allocated memory on success, 0 on failure.

void **devm_free_pages**(struct *device* * *dev*, unsigned long *addr*)
Resource-managed free_pages

Parameters

struct device * dev Device this memory belongs to

unsigned long addr Memory to free

Description

Free memory allocated with *devm_get_free_pages()*. Unlike free_pages, there is no need to supply the **order**.

void __percpu * **__devm_alloc_percpu**(struct *device* * *dev*, size_t *size*, size_t *align*)
Resource-managed alloc_percpu

Parameters

struct device * dev Device to allocate per-cpu memory for

size_t size Size of per-cpu memory to allocate

size_t align Alignment of per-cpu memory to allocate

Description

Managed alloc_percpu. Per-cpu memory allocated with this function is automatically freed on driver detach.

Return

Pointer to allocated memory on success, NULL on failure.

void **`devm_free_percpu`**(struct *device* * *dev*, void __percpu * *pdata*)
Resource-managed free_percpu

Parameters

`struct device * dev` Device this memory belongs to

`void __percpu * pdata` Per-cpu memory to free

Description

Free memory allocated with *`devm_alloc_percpu()`*.

CHAPTER

TWO

DEVICE DRIVERS INFRASTRUCTURE

The Basic Device Driver-Model Structures

struct **bus_type**
The bus type of the device

Definition

```
struct bus_type {
  const char              *name;
  const char              *dev_name;
  struct device           *dev_root;
  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);
  int (*num_vf)(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;
  bool force_dma;
};
```

Members

name The name of the bus.

dev_name Used for subsystems to enumerate devices like (“foo‘‘u‘‘”, dev->id).

dev_root Default device to use as the parent.

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 positive value if the given device can be handled by the given driver and zero otherwise. It may also return error code if determining that the driver supports the device is not possible. In case of -EPROBE_DEFER it will queue the device for deferred probing.

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.

num_vf Called to find out how many virtual functions a device on this bus supports.

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

force_dma Assume devices on this bus should be set up by `dma_configure()` even if DMA capability is not explicitly described by firmware.

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.

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.

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.

struct **device_driver**
The basic device driver structure

Definition

```
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;
  int (*coredump) (struct device *dev);
  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.

struct **subsys_interface**
interfaces to device functions

Definition

```
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 subsytem 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.

struct **class**
device classes

Definition

```
struct class {
  const char              *name;
  struct module           *owner;
  const struct attribute_group    **class_groups;
  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 (*shutdown_pre)(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_groups 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.

shutdown_pre Called at shut-down time before driver shutdown.

ns_type Callbacks so sysfs can detemine 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.

devm_alloc_percpu(*dev*, *type*)
Resource-managed alloc_percpu

Parameters

dev Device to allocate per-cpu memory for

type Type to allocate per-cpu memory for

Description

Managed alloc_percpu. Per-cpu memory allocated with this function is automatically freed on driver detach.

Return

Pointer to allocated memory on success, NULL on failure.

enum **device_link_state**
Device link states.

Constants

DL_STATE_NONE The presence of the drivers is not being tracked.

DL_STATE_DORMANT None of the supplier/consumer drivers is present.

DL_STATE_AVAILABLE The supplier driver is present, but the consumer is not.

DL_STATE_CONSUMER_PROBE The consumer is probing (supplier driver present).

DL_STATE_ACTIVE Both the supplier and consumer drivers are present.

DL_STATE_SUPPLIER_UNBIND The supplier driver is unbinding.

struct **device_link**
Device link representation.

Definition

```
struct device_link {
  struct device *supplier;
  struct list_head s_node;
  struct device *consumer;
  struct list_head c_node;
  enum device_link_state status;
  u32 flags;
  bool rpm_active;
#ifdef CONFIG_SRCU;
  struct rcu_head rcu_head;
#endif;
};
```

Members

supplier The device on the supplier end of the link.

s_node Hook to the supplier device’s list of links to consumers.

consumer The device on the consumer end of the link.

c_node Hook to the consumer device's list of links to suppliers.

status The state of the link (with respect to the presence of drivers).

flags Link flags.

rpm_active Whether or not the consumer device is runtime-PM-active.

rcu_head An RCU head to use for deferred execution of SRCU callbacks.

enum **dl_dev_state**
Device driver presence tracking information.

Constants

DL_DEV_NO_DRIVER There is no driver attached to the device.

DL_DEV_PROBING A driver is probing.

DL_DEV_DRIVER_BOUND The driver has been bound to the device.

DL_DEV_UNBINDING The driver is unbinding from the device.

struct **dev_links_info**
Device data related to device links.

Definition

```
struct dev_links_info {
  struct list_head suppliers;
  struct list_head consumers;
  enum dl_dev_state status;
};
```

Members

suppliers List of links to supplier devices.

consumers List of links to consumer devices.

status Driver status information.

struct **device**
The basic device structure

Definition

```
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_links_info   links;
  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;
  const struct dma_map_ops *dma_ops;
  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;
  struct iommu_fwspec     *iommu_fwspec;
  bool offline_disabled:1;
  bool offline:1;
  bool of_node_reused: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.

links Links to suppliers and consumers of this device.

power For device power management. See Documentation/driver-api/pm/devices.rst 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/driver-api/pinctl.rst for details.

msi_list Hosts MSI descriptors

numa_node NUMA node this device is close to.

dma_ops DMA mapping operations for this device.

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.

cma_area Contiguous memory area for dma allocations

archdata For arch-specific additions.

of_node Associated device tree node.

fwnode Associated device node supplied by platform firmware.

devt For creating the sysfs "dev".

id device instance

devres_lock Spinlock to protect the resource of the device.

devres_head The resources list of the device.

knode_class The node used to add the device to the class list.

class The class of the device.

groups Optional attribute groups.

release 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).

iommu_group IOMMU group the device belongs to.

iommu_fwspec IOMMU-specific properties supplied by firmware.

offline_disabled If set, the device is permanently online.

offline Set after successful invocation of bus type's .:c:func:*offline()*.

of_node_reused Set if the device-tree node is shared with an ancestor device.

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.

module_driver(*__driver*, *__register*, *__unregister*, *...*)
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()*.

Parameters

__driver driver name

__register register function for this driver type

__unregister unregister function for this driver type

... Additional arguments to be passed to __register and __unregister.

Description

Use this macro to construct bus specific macros for registering drivers, and do not use it on its own.

builtin_driver(*__driver*, *__register*, *...*)
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.

Parameters

__driver driver name

__register register function for this driver type

... Additional arguments to be passed to __register

Description

Use this macro to construct bus specific macros for registering drivers, and do not use it on its own.

Device Drivers Base

void **driver_init**(void)
initialize driver model.

Parameters

void no arguments

Description

Call the driver model init functions to initialize their subsystems. Called early from init/main.c.

int **driver_for_each_device**(struct *device_driver* * *drv*, struct *device* * *start*, void * *data*, int (*fn) (struct *device* *, void *))
Iterator for devices bound to a driver.

Parameters

struct device_driver * drv Driver we're iterating.

struct device * start Device to begin with

void * data Data to pass to the callback.

int (*)(struct device *, void *) fn Function to call for each device.

Description

Iterate over the **drv**'s list of devices calling **fn** for each one.

struct *device* * **driver_find_device**(struct *device_driver* * *drv*, struct *device* * *start*, void * *data*, int (*match) (struct *device* **dev*, void **data*))
device iterator for locating a particular device.

Parameters

`struct device_driver * drv` The device's driver

`struct device * start` Device to begin with

`void * data` Data to pass to match function

`int (*)(struct device *dev, void *data) 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.

int **`driver_create_file`**(struct *device_driver* * *drv*, const struct driver_attribute * *attr*)
create sysfs file for driver.

Parameters

`struct device_driver * drv` driver.

`const struct driver_attribute * attr` driver attribute descriptor.

void **`driver_remove_file`**(struct *device_driver* * *drv*, const struct driver_attribute * *attr*)
remove sysfs file for driver.

Parameters

`struct device_driver * drv` driver.

`const struct driver_attribute * attr` driver attribute descriptor.

int **`driver_register`**(struct *device_driver* * *drv*)
register driver with bus

Parameters

`struct device_driver * 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.

void **`driver_unregister`**(struct *device_driver* * *drv*)
remove driver from system.

Parameters

`struct device_driver * drv` driver.

Description

Again, we pass off most of the work to the bus-level call.

struct *device_driver* * **`driver_find`**(const char * *name*, struct *bus_type* * *bus*)
locate driver on a bus by its name.

Parameters

`const char * name` name of the driver.

`struct bus_type * 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.

struct *device_link* * **device_link_add**(struct *device* * *consumer*, struct *device* * *supplier*, u32 *flags*)
Create a link between two devices.

Parameters

struct device * consumer Consumer end of the link.

struct device * supplier Supplier end of the link.

u32 flags Link flags.

Description

The caller is responsible for the proper synchronization of the link creation with runtime PM. First, setting the DL_FLAG_PM_RUNTIME flag will cause the runtime PM framework to take the link into account. Second, if the DL_FLAG_RPM_ACTIVE flag is set in addition to it, the supplier devices will be forced into the active metastate and reference-counted upon the creation of the link. If DL_FLAG_PM_RUNTIME is not set, DL_FLAG_RPM_ACTIVE will be ignored.

If the DL_FLAG_AUTOREMOVE is set, the link will be removed automatically when the consumer device driver unbinds from it. The combination of both DL_FLAG_AUTOREMOVE and DL_FLAG_STATELESS set is invalid and will cause NULL to be returned.

A side effect of the link creation is re-ordering of dpm_list and the devices_kset list by moving the consumer device and all devices depending on it to the ends of these lists (that does not happen to devices that have not been registered when this function is called).

The supplier device is required to be registered when this function is called and NULL will be returned if that is not the case. The consumer device need not be registered, however.

void **device_link_del**(struct *device_link* * *link*)
Delete a link between two devices.

Parameters

struct device_link * link Device link to delete.

Description

The caller must ensure proper synchronization of this function with runtime PM.

const char * **dev_driver_string**(const struct *device* * *dev*)
Return a device's driver name, if at all possible

Parameters

const struct device * 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.

int **devm_device_add_group**(struct *device* * *dev*, const struct attribute_group * *grp*)
given a device, create a managed attribute group

Parameters

struct device * dev The device to create the group for

const struct attribute_group * grp The attribute group to create

Description

This function creates a group for the first time. It will explicitly warn and error if any of the attribute files being created already exist.

Returns 0 on success or error code on failure.

void **devm_device_remove_group**(struct *device* * *dev*, const struct attribute_group * *grp*)

Parameters

struct device * dev device to remove the group from

const struct attribute_group * grp group to remove

Description

This function removes a group of attributes from a device. The attributes previously have to have been created for this group, otherwise it will fail.

int **devm_device_add_groups**(struct *device* * *dev*, const struct attribute_group ** *groups*)
create a bunch of managed attribute groups

Parameters

struct device * dev The device to create the group for

const struct attribute_group ** groups The attribute groups to create, NULL terminated

Description

This function creates a bunch of managed attribute groups. If an error occurs when creating a group, all previously created groups will be removed, unwinding everything back to the original state when this function was called. It will explicitly warn and error if any of the attribute files being created already exist.

Returns 0 on success or error code from sysfs_create_group on failure.

void **devm_device_remove_groups**(struct *device* * *dev*, const struct attribute_group ** *groups*)
remove a list of managed groups

Parameters

struct device * dev The device for the groups to be removed from

const struct attribute_group ** groups NULL terminated list of groups to be removed

Description

If groups is not NULL, remove the specified groups from the device.

int **device_create_file**(struct *device* * *dev*, const struct device_attribute * *attr*)
create sysfs attribute file for device.

Parameters

struct device * dev device.

const struct device_attribute * attr device attribute descriptor.

void **device_remove_file**(struct *device* * *dev*, const struct device_attribute * *attr*)
remove sysfs attribute file.

Parameters

struct device * dev device.

const struct device_attribute * attr device attribute descriptor.

bool **device_remove_file_self**(struct *device* * *dev*, const struct device_attribute * *attr*)
remove sysfs attribute file from its own method.

Parameters

struct device * dev device.

const struct device_attribute * attr device attribute descriptor.

Description

See `kernfs_remove_self()` for details.

int **device_create_bin_file**(struct *device* * *dev*, const struct bin_attribute * *attr*)
create sysfs binary attribute file for device.

Parameters

`struct device * dev` device.

`const struct bin_attribute * attr` device binary attribute descriptor.

void **`device_remove_bin_file`**(struct *device* * *dev*, const struct bin_attribute * *attr*)
remove sysfs binary attribute file

Parameters

`struct device * dev` device.

`const struct bin_attribute * attr` device binary attribute descriptor.

void **`device_initialize`**(struct *device* * *dev*)
init device structure.

Parameters

`struct device * 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.

int **`dev_set_name`**(struct *device* * *dev*, const char * *fmt*, ...)
set a device name

Parameters

`struct device * dev` device

`const char * fmt` format string for the device's name

`...` variable arguments

int **`device_add`**(struct *device* * *dev*)
add device to device hierarchy.

Parameters

`struct device * 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.

int **device_register**(struct *device* * *dev*)
register a device with the system.

Parameters

struct device * 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.

struct *device* * **get_device**(struct *device* * *dev*)
increment reference count for device.

Parameters

struct device * 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.

void **put_device**(struct *device* * *dev*)
decrement reference count.

Parameters

struct device * dev device in question.

void **device_del**(struct *device* * *dev*)
delete device from system.

Parameters

struct device * 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.

void **device_unregister**(struct *device* * *dev*)
unregister device from system.

Parameters

struct device * 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.

int **device_for_each_child**(struct *device* * *parent*, void * *data*, int (*fn) (struct *device* **dev*, void **data*)
device child iterator.

Parameters

struct device * parent parent struct device.

void * data data for the callback.

int (*)(struct device *dev, void *data) 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.

int **device_for_each_child_reverse**(struct *device* * *parent*, void * *data*, int (*fn) (struct *device* **dev*, void **data*)
device child iterator in reversed order.

Parameters

struct device * parent parent struct device.

void * data data for the callback.

int (*)(struct device *dev, void *data) 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.

struct *device* * **device_find_child**(struct *device* * *parent*, void * *data*, int (*match) (struct *device* **dev*, void **data*)
device iterator for locating a particular device.

Parameters

struct device * parent parent struct device

void * data Data to pass to match function

int (*)(struct device *dev, void *data) 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.

struct *device* * **__root_device_register**(const char * *name*, struct module * *owner*)
allocate and register a root device

Parameters

const char * name root device name

struct module * 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().

void **root_device_unregister**(struct *device* * *dev*)
unregister and free a root device

Parameters

struct device * dev device going away

Description

This function unregisters and cleans up a device that was created by root_device_register().

struct *device* * **device_create_vargs**(struct *class* * *class*, struct *device* * *parent*, dev_t *devt*, void * *drvdata*, const char * *fmt*, va_list *args*)
creates a device and registers it with sysfs

Parameters

struct class * class pointer to the struct class that this device should be registered to

struct device * parent pointer to the parent struct device of this new device, if any

dev_t devt the dev_t for the char device to be added

void * drvdata the data to be added to the device for callbacks

const char * fmt string for the device's name

va_list args 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().

struct *device* * **device_create**(struct *class* * *class*, struct *device* * *parent*, dev_t *devt*, void * *drvdata*, const char * *fmt*, ...)
creates a device and registers it with sysfs

Parameters

struct class * class pointer to the struct class that this device should be registered to

struct device * parent pointer to the parent struct device of this new device, if any

dev_t devt the dev_t for the char device to be added

void * drvdata the data to be added to the device for callbacks

const char * fmt 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()`.

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*, ...)

creates a device and registers it with sysfs

Parameters

struct class * class pointer to the struct class that this device should be registered to

struct device * parent pointer to the parent struct device of this new device, if any

dev_t devt the dev_t for the char device to be added

void * drvdata the data to be added to the device for callbacks

const struct attribute_group ** groups NULL-terminated list of attribute groups to be created

const char * fmt 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()`.

void **device_destroy**(struct *class* * *class*, dev_t *devt*)

removes a device that was created with *device_create()*

Parameters

struct class * class pointer to the struct class that this device was registered with

dev_t 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()*.

int **device_rename**(struct *device* * *dev*, const char * *new_name*)
renames a device

Parameters

struct device * dev the pointer to the struct device to be renamed

const char * 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. :)

int **device_move**(struct *device* * *dev*, struct *device* * *new_parent*, enum dpm_order *dpm_order*)
moves a device to a new parent

Parameters

struct device * dev the pointer to the struct device to be moved

struct device * new_parent the new parent of the device (can by NULL)

enum dpm_order dpm_order how to reorder the dpm_list

void **set_primary_fwnode**(struct *device* * *dev*, struct fwnode_handle * *fwnode*)
Change the primary firmware node of a given device.

Parameters

struct device * dev Device to handle.

struct fwnode_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.

void **device_set_of_node_from_dev**(struct *device* * *dev*, const struct *device* * *dev2*)
reuse device-tree node of another device

Parameters

struct device * dev device whose device-tree node is being set

const struct device * dev2 device whose device-tree node is being reused

Description

Takes another reference to the new device-tree node after first dropping any reference held to the old node.

void **register_syscore_ops**(struct syscore_ops * *ops*)
Register a set of system core operations.

Parameters

struct syscore_ops * ops System core operations to register.

void **unregister_syscore_ops**(struct syscore_ops * *ops*)
Unregister a set of system core operations.

Parameters

struct syscore_ops * ops System core operations to unregister.

int **syscore_suspend**(void)
Execute all the registered system core suspend callbacks.

Parameters

void no arguments

Description

This function is executed with one CPU on-line and disabled interrupts.

void **syscore_resume**(void)
Execute all the registered system core resume callbacks.

Parameters

void no arguments

Description

This function is executed with one CPU on-line and disabled interrupts.

struct *class* * **__class_create**(struct module * *owner*, const char * *name*, struct lock_class_key * *key*)
create a struct class structure

Parameters

struct module * owner pointer to the module that is to "own" this struct class

const char * name pointer to a string for the name of this class.

struct lock_class_key * 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()*.

void **class_destroy**(struct *class* * *cls*)
destroys a struct class structure

Parameters

struct class * 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()`.

void **class_dev_iter_init**(struct class_dev_iter * *iter*, struct *class* * *class*, struct *device* * *start*, const struct device_type * *type*)
initialize class device iterator

Parameters

struct class_dev_iter * iter class iterator to initialize

struct class * class the class we wanna iterate over

struct device * start the device to start iterating from, if any

const struct device_type * 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.

struct *device* * **class_dev_iter_next**(struct class_dev_iter * *iter*)
iterate to the next device

Parameters

struct class_dev_iter * 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.

void **class_dev_iter_exit**(struct class_dev_iter * *iter*)
finish iteration

Parameters

struct class_dev_iter * 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.

int **class_for_each_device**(struct *class* * *class*, struct *device* * *start*, void * *data*, int (*fn) (struct *device* *, void *)
device iterator

Parameters

struct class * class the class we're iterating

struct device * start the device to start with in the list, if any.

void * data data for the callback

int (*)(struct device *, void *) 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.

struct *device* * **class_find_device**(struct *class* * *class*, struct *device* * *start*, const void * *data*, int (*match) (struct *device* *, const void *))
device iterator for locating a particular device

Parameters

struct class * class the class we're iterating

struct device * start Device to begin with

const void * data data for the match function

int (*)(struct device *, const void *) 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.

struct class_compat * **class_compat_register**(const char * *name*)
register a compatibility class

Parameters

const char * 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.

void **class_compat_unregister**(struct class_compat * *cls*)
unregister a compatibility class

Parameters

struct class_compat * cls the class to unregister

int **class_compat_create_link**(struct class_compat * *cls*, struct *device* * *dev*, struct *device* * *device_link*)
create a compatibility class device link to a bus device

Parameters

struct class_compat * cls the compatibility class

struct device * dev the target bus device

struct device * device_link an optional device to which a "device" link should be created

void **class_compat_remove_link**(struct class_compat * *cls*, struct *device* * *dev*, struct *device* * *device_link*)
remove a compatibility class device link to a bus device

Parameters

struct class_compat * cls the compatibility class

struct device * dev the target bus device

struct device * device_link an optional device to which a "device" link was previously created

void **unregister_node**(struct node * *node*)
unregister a node device

Parameters

struct node * node node going away

Description

Unregisters a node device **node**. All the devices on the node must be unregistered before calling this function.

int **request_firmware**(const struct firmware ** *firmware_p*, const char * *name*, struct device * *device*)
send firmware request and wait for it

Parameters

const struct firmware ** firmware_p pointer to firmware image

const char * name name of firmware file

struct device * device 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.

int **request_firmware_direct**(const struct firmware ** *firmware_p*, const char * *name*, struct device * *device*)
load firmware directly without usermode helper

Parameters

const struct firmware ** firmware_p pointer to firmware image

const char * name name of firmware file

struct device * device 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.

int **request_firmware_into_buf**(const struct firmware ** *firmware_p*, const char * *name*, struct device * *device*, void * *buf*, size_t *size*)
load firmware into a previously allocated buffer

Parameters

const struct firmware ** firmware_p pointer to firmware image

const char * name name of firmware file

struct device * device device for which firmware is being loaded and DMA region allocated

void * buf address of buffer to load firmware into

size_t size size of buffer

Description

This function works pretty much like *request_firmware()*, but it doesn't allocate a buffer to hold the firmware data. Instead, the firmware is loaded directly into the buffer pointed to by **buf** and the **firmware_p** data member is pointed at **buf**.

This function doesn't cache firmware either.

void **release_firmware**(const struct firmware * *fw*)
release the resource associated with a firmware image

Parameters

const struct firmware * fw firmware resource to release

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*))
asynchronous version of request_firmware

Parameters

struct module * module module requesting the firmware

bool uevent sends uevent to copy the firmware image if this flag is non-zero else the firmware copy must be done manually.

const char * name name of firmware file

struct device * device device for which firmware is being loaded

gfp_t gfp allocation flags

void * context will be passed over to **cont**, and **fw** may be NULL if firmware request fails.

void (*)(const struct firmware *fw, void *context) cont 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 ->:c:func:*probe()* methods, if **gfp** is GFP_KERNEL.
- can't sleep at all if **gfp** is GFP_ATOMIC.

int **transport_class_register**(struct transport_class * *tclass*)
register an initial transport class

Parameters

struct transport_class * 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.

void **transport_class_unregister**(struct transport_class * *tclass*)
unregister a previously registered class

Parameters

struct transport_class * tclass The transport class to unregister

Description

Must be called prior to deallocating the memory for the transport class.

int **anon_transport_class_register**(struct anon_transport_class * *atc*)
register an anonymous class

Parameters

struct anon_transport_class * 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.

void **anon_transport_class_unregister**(struct anon_transport_class * *atc*)
unregister an anon class

Parameters

struct anon_transport_class * atc Pointer to the anon transport class to unregister

Description

Must be called prior to deallocating the memory for the anon transport class.

void **transport_setup_device**(struct *device* * *dev*)
declare a new dev for transport class association but don't make it visible yet.

Parameters

struct device * 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).

void **transport_add_device**(struct *device* * *dev*)
declare a new dev for transport class association

Parameters

struct device * 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.

void **transport_configure_device**(struct *device* * *dev*)
configure an already set up device

Parameters

struct device * 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.

void **transport_remove_device**(struct *device* * *dev*)
remove the visibility of a device

Parameters

struct device * 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.

void **transport_destroy_device**(struct *device* * *dev*)
destroy a removed device

Parameters

struct device * 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.

int **device_bind_driver**(struct *device* * *dev*)
bind a driver to one device.

Parameters

struct device * 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.

void **wait_for_device_probe**(void)

Parameters

void no arguments

Description

Wait for device probing to be completed.

int **device_attach**(struct *device* * *dev*)
try to attach device to a driver.

Parameters

struct device * 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.

int **driver_attach**(struct *device_driver* * *drv*)
try to bind driver to devices.

Parameters

struct device_driver * 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.

void **device_release_driver**(struct *device* * *dev*)
manually detach device from driver.

Parameters

struct device * dev device.

Description

Manually detach device from driver. When called for a USB interface, **dev**->parent lock must be held.

If this function is to be called with **dev**->parent lock held, ensure that the device's consumers are unbound in advance or that their locks can be acquired under the **dev**->parent lock.

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*)
add a platform-level device with resources and platform-specific data

Parameters

struct device * parent parent device for the device we're adding

const char * name base name of the device we're adding

int id instance id

const struct resource * res set of resources that needs to be allocated for the device

unsigned int num number of resources

const void * data platform specific data for this platform device

size_t size size of platform specific data

Description

Returns `struct platform_device` pointer on success, or ERR_PTR() on error.

struct platform_device * **platform_device_register_simple**(const char * *name*, int *id*, const struct resource * *res*, unsigned int *num*)
add a platform-level device and its resources

Parameters

const char * name base name of the device we're adding

int id instance id

const struct resource * res set of resources that needs to be allocated for the device

unsigned int 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.

struct platform_device * **platform_device_register_data**(struct *device* * *parent*, const char * *name*, int *id*, const void * *data*, size_t *size*)
add a platform-level device with platform-specific data

Parameters

struct device * parent parent device for the device we’re adding

const char * name base name of the device we’re adding

int id instance id

const void * data platform specific data for this platform device

size_t size 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.

struct resource * **platform_get_resource**(struct platform_device * *dev*, unsigned int *type*, unsigned int *num*)
get a resource for a device

Parameters

struct platform_device * dev platform device

unsigned int type resource type

unsigned int num resource index

int **platform_get_irq**(struct platform_device * *dev*, unsigned int *num*)
get an IRQ for a device

Parameters

struct platform_device * dev platform device

unsigned int num IRQ number index

int **platform_irq_count**(struct platform_device * *dev*)
Count the number of IRQs a platform device uses

Parameters

struct platform_device * dev platform device

Return

Number of IRQs a platform device uses or EPROBE_DEFER

struct resource * **platform_get_resource_byname**(struct platform_device * *dev*, unsigned int *type*, const char * *name*)
get a resource for a device by name

Parameters

struct platform_device * dev platform device

unsigned int type resource type

const char * name resource name

int **platform_get_irq_byname**(struct platform_device * *dev*, const char * *name*)
get an IRQ for a device by name

Parameters

`struct platform_device * dev` platform device

`const char * name` IRQ name

int **`platform_add_devices`**(struct platform_device ** *devs*, int *num*)
add a numbers of platform devices

Parameters

`struct platform_device ** devs` array of platform devices to add

`int num` number of platform devices in array

void **`platform_device_put`**(struct platform_device * *pdev*)
destroy a platform device

Parameters

`struct platform_device * 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.

struct platform_device * **`platform_device_alloc`**(const char * *name*, int *id*)
create a platform device

Parameters

`const char * name` base name of the device we're adding

`int 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.

int **`platform_device_add_resources`**(struct platform_device * *pdev*, const struct resource * *res*, unsigned int *num*)
add resources to a platform device

Parameters

`struct platform_device * pdev` platform device allocated by platform_device_alloc to add resources to

`const struct resource * res` set of resources that needs to be allocated for the device

`unsigned int 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.

int **`platform_device_add_data`**(struct platform_device * *pdev*, const void * *data*, size_t *size*)
add platform-specific data to a platform device

Parameters

`struct platform_device * pdev` platform device allocated by platform_device_alloc to add resources to

`const void * data` platform specific data for this platform device

`size_t 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.

int **platform_device_add_properties**(struct platform_device * *pdev*, const struct property_entry * *properties*)
add built-in properties to a platform device

Parameters

struct platform_device * pdev platform device to add properties to

const struct property_entry * properties null terminated array of properties to add

Description

The function will take deep copy of **properties** and attach the copy to the platform device. The memory associated with properties will be freed when the platform device is released.

int **platform_device_add**(struct platform_device * *pdev*)
add a platform device to device hierarchy

Parameters

struct platform_device * 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()*.

void **platform_device_del**(struct platform_device * *pdev*)
remove a platform-level device

Parameters

struct platform_device * 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.

int **platform_device_register**(struct platform_device * *pdev*)
add a platform-level device

Parameters

struct platform_device * pdev platform device we're adding

void **platform_device_unregister**(struct platform_device * *pdev*)
unregister a platform-level device

Parameters

struct platform_device * 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()*.

struct platform_device * **platform_device_register_full**(const struct platform_device_info * *pdevinfo*)
add a platform-level device with resources and platform-specific data

Parameters

const struct platform_device_info * pdevinfo data used to create device

Description

Returns `struct platform_device` pointer on success, or ERR_PTR() on error.

int **__platform_driver_register**(struct platform_driver * *drv*, struct module * *owner*)
register a driver for platform-level devices

Parameters

struct platform_driver * drv platform driver structure

struct module * owner owning module/driver

void **platform_driver_unregister**(struct platform_driver * *drv*)
unregister a driver for platform-level devices

Parameters

struct platform_driver * drv platform driver structure

int **__platform_driver_probe**(struct platform_driver * *drv*, int (*probe) (struct platform_device *, struct module * *module*)
register driver for non-hotpluggable device

Parameters

struct platform_driver * drv platform driver structure

int (*)(struct platform_device *) probe the driver probe routine, probably from an __init section

struct module * 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.

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*)
register driver and create corresponding device

Parameters

struct platform_driver * driver platform driver structure

int (*)(struct platform_device *) probe the driver probe routine, probably from an __init section

struct resource * res set of resources that needs to be allocated for the device

unsigned int n_res number of resources

const void * data platform specific data for this platform device

size_t size size of platform specific data

struct module * 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.

int **__platform_register_drivers**(struct platform_driver *const * *drivers*, unsigned int *count*, struct module * *owner*)
register an array of platform drivers

Parameters

struct platform_driver *const * drivers an array of drivers to register

unsigned int count the number of drivers to register

struct module * 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.

Return

0 on success or a negative error code on failure.

void **platform_unregister_drivers**(struct platform_driver *const * *drivers*, unsigned int *count*)
unregister an array of platform drivers

Parameters

struct platform_driver *const * drivers an array of drivers to unregister

unsigned int count the number of drivers to unregister

Description

Unegisters 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.

int **bus_for_each_dev**(struct *bus_type* * *bus*, struct *device* * *start*, void * *data*, int (*fn) (struct *device* *, void *)
device iterator.

Parameters

struct bus_type * bus bus type.

struct device * start device to start iterating from.

void * data data for the callback.

int (*)(struct device *, void *) 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.

struct *device* * **bus_find_device**(struct *bus_type* * *bus*, struct *device* * *start*, void * *data*, int (*match) (struct *device* **dev*, void **data*)
device iterator for locating a particular device.

Parameters

struct bus_type * bus bus type

struct device * start Device to begin with

void * data Data to pass to match function

int (*)(struct device *dev, void *data) 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.

struct *device* * **bus_find_device_by_name**(struct *bus_type* * *bus*, struct *device* * *start*, const char * *name*)
device iterator for locating a particular device of a specific name

Parameters

struct bus_type * bus bus type

struct device * start Device to begin with

const char * 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.

struct *device* * **subsys_find_device_by_id**(struct *bus_type* * *subsys*, unsigned int *id*, struct *device* * *hint*)
find a device with a specific enumeration number

Parameters

struct bus_type * subsys subsystem

unsigned int id index 'id' in struct device

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.

int **bus_for_each_drv**(struct *bus_type* * *bus*, struct *device_driver* * *start*, void * *data*, int (*fn) (struct *device_driver* *, void *)
driver iterator

Parameters

struct bus_type * bus bus we're dealing with.

struct device_driver * start driver to start iterating on.

void * data data to pass to the callback.

int (*)(struct device_driver *, void *) 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.

int **bus_rescan_devices**(struct *bus_type* * *bus*)
rescan devices on the bus for possible drivers

Parameters

struct bus_type * 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.

int **device_reprobe**(struct *device* * *dev*)
remove driver for a device and probe for a new driver

Parameters

struct device * 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.

int **bus_register**(struct *bus_type* * *bus*)
register a driver-core subsystem

Parameters

struct bus_type * 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.

void **bus_unregister**(struct *bus_type* * *bus*)
remove a bus from the system

Parameters

struct bus_type * bus bus.

Description

Unregister the child subsystems and the bus itself. Finally, we call `bus_put()` to release the refcount

void **subsys_dev_iter_init**(struct subsys_dev_iter * *iter*, struct *bus_type* * *subsys*, struct *device* * *start*, const struct device_type * *type*)
initialize subsys device iterator

Parameters

struct subsys_dev_iter * iter subsys iterator to initialize

struct bus_type * subsys the subsys we wanna iterate over

struct device * start the device to start iterating from, if any

const struct device_type * 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.

struct *device* * **subsys_dev_iter_next**(struct subsys_dev_iter * *iter*)
iterate to the next device

Parameters

struct subsys_dev_iter * 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.

void **subsys_dev_iter_exit**(struct subsys_dev_iter * *iter*)
finish iteration

Parameters

struct subsys_dev_iter * 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.

int **subsys_system_register**(struct *bus_type* * *subsys*, const struct attribute_group ** *groups*)
register a subsystem at /sys/devices/system/

Parameters

struct bus_type * subsys system subsystem

const struct attribute_group ** 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>.

int **subsys_virtual_register**(struct *bus_type* * *subsys*, const struct attribute_group ** *groups*)
register a subsystem at /sys/devices/virtual/

Parameters

struct bus_type * subsys virtual subsystem

const struct attribute_group ** groups default attributes for the root device

Description

All 'virtual' subsystems have a /sys/devices/system/<name> root device with the name of the subystem. 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

int **dma_alloc_from_dev_coherent**(struct *device* * *dev*, ssize_t *size*, dma_addr_t * *dma_handle*, void ** *ret*)
allocate memory from device coherent pool

Parameters

struct device * dev device from which we allocate memory

ssize_t size size of requested memory area

dma_addr_t * dma_handle This will be filled with the correct dma handle

void ** ret 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**.

int **`dma_release_from_dev_coherent`**(struct *device* * *dev*, int *order*, void * *vaddr*)
free memory to device coherent memory pool

Parameters

`struct device * dev` device from which the memory was allocated

`int order` the order of pages allocated

`void * 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 the caller should proceed with releasing memory from generic pools.

int **`dma_mmap_from_dev_coherent`**(struct *device* * *dev*, struct vm_area_struct * *vma*, void * *vaddr*, size_t *size*, int * *ret*)
mmap memory from the device coherent pool

Parameters

`struct device * dev` device from which the memory was allocated

`struct vm_area_struct * vma` vm_area for the userspace memory

`void * vaddr` cpu address returned by dma_alloc_from_dev_coherent

`size_t size` size of the memory buffer allocated

`int * ret` result from `remap_pfn_range()`

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.

void * **`dmam_alloc_coherent`**(struct *device* * *dev*, size_t *size*, dma_addr_t * *dma_handle*, gfp_t *gfp*)
Managed `dma_alloc_coherent()`

Parameters

`struct device * dev` Device to allocate coherent memory for

`size_t size` Size of allocation

`dma_addr_t * dma_handle` Out argument for allocated DMA handle

`gfp_t gfp` Allocation flags

Description

Managed `dma_alloc_coherent()`. Memory allocated using this function will be automatically released on driver detach.

Return

Pointer to allocated memory on success, NULL on failure.

void **dmam_free_coherent**(struct *device* * *dev*, size_t *size*, void * *vaddr*, dma_addr_t *dma_handle*)
Managed `dma_free_coherent()`

Parameters

`struct device * dev` Device to free coherent memory for

`size_t size` Size of allocation

`void * vaddr` Virtual address of the memory to free

`dma_addr_t dma_handle` DMA handle of the memory to free

Description

Managed `dma_free_coherent()`.

void * **dmam_alloc_attrs**(struct *device* * *dev*, size_t *size*, dma_addr_t * *dma_handle*, gfp_t *gfp*, unsigned long *attrs*)
Managed `dma_alloc_attrs()`

Parameters

`struct device * dev` Device to allocate non_coherent memory for

`size_t size` Size of allocation

`dma_addr_t * dma_handle` Out argument for allocated DMA handle

`gfp_t gfp` Allocation flags

`unsigned long attrs` Flags in the DMA_ATTR_* namespace.

Description

Managed `dma_alloc_attrs()`. Memory allocated using this function will be automatically released on driver detach.

Return

Pointer to allocated memory on success, NULL on failure.

int **dmam_declare_coherent_memory**(struct *device* * *dev*, phys_addr_t *phys_addr*, dma_addr_t *device_addr*, size_t *size*, int *flags*)
Managed `dma_declare_coherent_memory()`

Parameters

`struct device * dev` Device to declare coherent memory for

`phys_addr_t phys_addr` Physical address of coherent memory to be declared

`dma_addr_t device_addr` Device address of coherent memory to be declared

`size_t size` Size of coherent memory to be declared

`int flags` Flags

Description

Managed `dma_declare_coherent_memory()`.

Return

0 on success, -errno on failure.

void **dmam_release_declared_memory**(struct *device* * *dev*)
Managed `dma_release_declared_memory()`.

Parameters

`struct device * dev` Device to release declared coherent memory for

Description

Managed *dmam_release_declared_memory()*.

Device drivers PnP support

int **pnp_register_protocol**(struct pnp_protocol * *protocol*)
adds a pnp protocol to the pnp layer

Parameters

struct pnp_protocol * protocol pointer to the corresponding pnp_protocol structure

Description

Ex protocols: ISAPNP, PNPBIOS, etc

void **pnp_unregister_protocol**(struct pnp_protocol * *protocol*)
removes a pnp protocol from the pnp layer

Parameters

struct pnp_protocol * protocol pointer to the corresponding pnp_protocol structure

struct pnp_dev * **pnp_request_card_device**(struct pnp_card_link * *clink*, const char * *id*, struct pnp_dev * *from*)
Searches for a PnP device under the specified card

Parameters

struct pnp_card_link * clink pointer to the card link, cannot be NULL

const char * id pointer to a PnP ID structure that explains the rules for finding the device

struct pnp_dev * from Starting place to search from. If NULL it will start from the beginning.

void **pnp_release_card_device**(struct pnp_dev * *dev*)
call this when the driver no longer needs the device

Parameters

struct pnp_dev * dev pointer to the PnP device structure

int **pnp_register_card_driver**(struct pnp_card_driver * *drv*)
registers a PnP card driver with the PnP Layer

Parameters

struct pnp_card_driver * drv pointer to the driver to register

void **pnp_unregister_card_driver**(struct pnp_card_driver * *drv*)
unregisters a PnP card driver from the PnP Layer

Parameters

struct pnp_card_driver * drv pointer to the driver to unregister

struct pnp_id * **pnp_add_id**(struct pnp_dev * *dev*, const char * *id*)
adds an EISA id to the specified device

Parameters

struct pnp_dev * dev pointer to the desired device

const char * id pointer to an EISA id string

int **pnp_start_dev**(struct pnp_dev * *dev*)
low-level start of the PnP device

Parameters

struct pnp_dev * dev pointer to the desired device

Description

assumes that resources have already been allocated

int **pnp_stop_dev**(struct pnp_dev * *dev*)
    low-level disable of the PnP device

Parameters

struct pnp_dev * dev pointer to the desired device

Description

does not free resources

int **pnp_activate_dev**(struct pnp_dev * *dev*)
    activates a PnP device for use

Parameters

struct pnp_dev * dev pointer to the desired device

Description

does not validate or set resources so be careful.

int **pnp_disable_dev**(struct pnp_dev * *dev*)
    disables device

Parameters

struct pnp_dev * dev pointer to the desired device

Description

inform the correct pnp protocol so that resources can be used by other devices

int **pnp_is_active**(struct pnp_dev * *dev*)
    Determines if a device is active based on its current resources

Parameters

struct pnp_dev * dev pointer to the desired PnP device

Userspace IO devices

void **uio_event_notify**(struct uio_info * *info*)
    trigger an interrupt event

Parameters

struct uio_info * info UIO device capabilities

int **__uio_register_device**(struct module * *owner*, struct device * *parent*, struct uio_info * *info*)
    register a new userspace IO device

Parameters

struct module * owner module that creates the new device

struct device * parent parent device

struct uio_info * info UIO device capabilities

Description

returns zero on success or a negative error code.

void **uio_unregister_device**(struct *uio_info* * *info*)
unregister a industrial IO device

Parameters

struct uio_info * info UIO device capabilities

struct **uio_mem**
description of a UIO memory region

Definition

```
struct uio_mem {
  const char              *name;
  phys_addr_t addr;
  unsigned long           offs;
  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 rounded to page size (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)

offs offset of device memory within the page

size size of IO (multiple of page size)

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.

struct **uio_port**
description of a UIO port region

Definition

```
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.

struct **uio_info**
UIO device capabilities

Definition

```
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 list of mappable memory regions, size==0 for end of list

port 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

THREE

DEVICE POWER MANAGEMENT

Device Power Management Basics

Most of the code in Linux is device drivers, so most of the Linux power management (PM) code is also driver-specific. Most drivers will do very little; others, especially for platforms with small batteries (like cell phones), will do a lot.

This writeup gives an overview of how drivers interact with system-wide power management goals, emphasizing the models and interfaces that are shared by everything that hooks up to the driver model core. Read it as background for the domain-specific work you'd do with any specific driver.

Two Models for Device Power Management

Drivers will use one or both of these models to put devices into low-power states:

System Sleep model:

> Drivers can enter low-power states as part of entering system-wide low-power states like "suspend" (also known as "suspend-to-RAM"), or (mostly for systems with disks) "hibernation" (also known as "suspend-to-disk").
>
> This is something that device, bus, and class drivers collaborate on by implementing various role-specific suspend and resume methods to cleanly power down hardware and software subsystems, then reactivate them without loss of data.
>
> Some drivers can manage hardware wakeup events, which make the system leave the low-power state. This feature may be enabled or disabled using the relevant `/sys/devices/.../power/wakeup` file (for Ethernet drivers the ioctl interface used by ethtool may also be used for this purpose); enabling it may cost some power usage, but let the whole system enter low-power states more often.

Runtime Power Management model:

> Devices may also be put into low-power states while the system is running, independently of other power management activity in principle. However, devices are not generally independent of each other (for example, a parent device cannot be suspended unless all of its child devices have been suspended). Moreover, depending on the bus type the device is on, it may be necessary to carry out some bus-specific operations on the device for this purpose. Devices put into low power states at run time may require special handling during system-wide power transitions (suspend or hibernation).
>
> For these reasons not only the device driver itself, but also the appropriate subsystem (bus type, device type or device class) driver and the PM core are involved in runtime power management. As in the system sleep power management case, they need to collaborate by implementing various role-specific suspend and resume methods, so

that the hardware is cleanly powered down and reactivated without data or service loss.

There's not a lot to be said about those low-power states except that they are very system-specific, and often device-specific. Also, that if enough devices have been put into low-power states (at runtime), the effect may be very similar to entering some system-wide low-power state (system sleep) … and that synergies exist, so that several drivers using runtime PM might put the system into a state where even deeper power saving options are available.

Most suspended devices will have quiesced all I/O: no more DMA or IRQs (except for wakeup events), no more data read or written, and requests from upstream drivers are no longer accepted. A given bus or platform may have different requirements though.

Examples of hardware wakeup events include an alarm from a real time clock, network wake-on-LAN packets, keyboard or mouse activity, and media insertion or removal (for PCMCIA, MMC/SD, USB, and so on).

Interfaces for Entering System Sleep States

There are programming interfaces provided for subsystems (bus type, device type, device class) and device drivers to allow them to participate in the power management of devices they are concerned with. These interfaces cover both system sleep and runtime power management.

Device Power Management Operations

Device power management operations, at the subsystem level as well as at the device driver level, are implemented by defining and populating objects of type *`struct dev_pm_ops`* defined in `include/linux/pm.h`. The roles of the methods included in it will be explained in what follows. For now, it should be sufficient to remember that the last three methods are specific to runtime power management while the remaining ones are used during system-wide power transitions.

There also is a deprecated "old" or "legacy" interface for power management operations available at least for some subsystems. This approach does not use *`struct dev_pm_ops`* objects and it is suitable only for implementing system sleep power management methods in a limited way. Therefore it is not described in this document, so please refer directly to the source code for more information about it.

Subsystem-Level Methods

The core methods to suspend and resume devices reside in *`struct dev_pm_ops`* pointed to by the ops member of *`struct dev_pm_domain`*, or by the pm member of *`struct bus_type`*, `struct device_type` and *`struct class`*. They are mostly of interest to the people writing infrastructure for platforms and buses, like PCI or USB, or device type and device class drivers. They also are relevant to the writers of device drivers whose subsystems (PM domains, device types, device classes and bus types) don't provide all power management methods.

Bus drivers implement these methods as appropriate for the hardware and the drivers using it; PCI works differently from USB, and so on. Not many people write subsystem-level drivers; most driver code is a "device driver" that builds on top of bus-specific framework code.

For more information on these driver calls, see the description later; they are called in phases for every device, respecting the parent-child sequencing in the driver model tree.

`/sys/devices/.../power/wakeup` files

All device objects in the driver model contain fields that control the handling of system wakeup events (hardware signals that can force the system out of a sleep state). These fields are initialized by bus or device driver code using `device_set_wakeup_capable()` and `device_set_wakeup_enable()`, defined in `include/linux/pm_wakeup.h`.

The `power.can_wakeup` flag just records whether the device (and its driver) can physically support wakeup events. The `device_set_wakeup_capable()` routine affects this flag. The `power.wakeup` field is a pointer to an object of type `struct wakeup_source` used for controlling whether or not the device should use its system wakeup mechanism and for notifying the PM core of system wakeup events signaled by the device. This object is only present for wakeup-capable devices (i.e. devices whose can_wakeup flags are set) and is created (or removed) by `device_set_wakeup_capable()`.

Whether or not a device is capable of issuing wakeup events is a hardware matter, and the kernel is responsible for keeping track of it. By contrast, whether or not a wakeup-capable device should issue wakeup events is a policy decision, and it is managed by user space through a sysfs attribute: the `power/wakeup` file. User space can write the "enabled" or "disabled" strings to it to indicate whether or not, respectively, the device is supposed to signal system wakeup. This file is only present if the `power.wakeup` object exists for the given device and is created (or removed) along with that object, by `device_set_wakeup_capable()`. Reads from the file will return the corresponding string.

The initial value in the `power/wakeup` file is "disabled" for the majority of devices; the major exceptions are power buttons, keyboards, and Ethernet adapters whose WoL (wake-on-LAN) feature has been set up with ethtool. It should also default to "enabled" for devices that don't generate wakeup requests on their own but merely forward wakeup requests from one bus to another (like PCI Express ports).

The `device_may_wakeup()` routine returns true only if the `power.wakeup` object exists and the corresponding `power/wakeup` file contains the "enabled" string. This information is used by subsystems, like the PCI bus type code, to see whether or not to enable the devices' wakeup mechanisms. If device wakeup mechanisms are enabled or disabled directly by drivers, they also should use `device_may_wakeup()` to decide what to do during a system sleep transition. Device drivers, however, are not expected to call `device_set_wakeup_enable()` directly in any case.

It ought to be noted that system wakeup is conceptually different from "remote wakeup" used by runtime power management, although it may be supported by the same physical mechanism. Remote wakeup is a feature allowing devices in low-power states to trigger specific interrupts to signal conditions in which they should be put into the full-power state. Those interrupts may or may not be used to signal system wakeup events, depending on the hardware design. On some systems it is impossible to trigger them from system sleep states. In any case, remote wakeup should always be enabled for runtime power management for all devices and drivers that support it.

`/sys/devices/.../power/control` files

Each device in the driver model has a flag to control whether it is subject to runtime power management. This flag, `runtime_auto`, is initialized by the bus type (or generally subsystem) code using `pm_runtime_allow()` or `pm_runtime_forbid()`; the default is to allow runtime power management.

The setting can be adjusted by user space by writing either "on" or "auto" to the device's `power/control` sysfs file. Writing "auto" calls `pm_runtime_allow()`, setting the flag and allowing the device to be runtime power-managed by its driver. Writing "on" calls `pm_runtime_forbid()`, clearing the flag, returning the device to full power if it was in a low-power state, and preventing the device from being runtime power-managed. User space can check the current value of the `runtime_auto` flag by reading that file.

The device's `runtime_auto` flag has no effect on the handling of system-wide power transitions. In particular, the device can (and in the majority of cases should and will) be put into a low-power state during a system-wide transition to a sleep state even though its `runtime_auto` flag is clear.

For more information about the runtime power management framework, refer to `Documentation/power/runtime_pm.txt`.

Calling Drivers to Enter and Leave System Sleep States

When the system goes into a sleep state, each device's driver is asked to suspend the device by putting it into a state compatible with the target system state. That's usually some version of "off", but the details are system-specific. Also, wakeup-enabled devices will usually stay partly functional in order to wake the system.

When the system leaves that low-power state, the device's driver is asked to resume it by returning it to full power. The suspend and resume operations always go together, and both are multi-phase operations.

For simple drivers, suspend might quiesce the device using class code and then turn its hardware as "off" as possible during suspend_noirq. The matching resume calls would then completely reinitialize the hardware before reactivating its class I/O queues.

More power-aware drivers might prepare the devices for triggering system wakeup events.

Call Sequence Guarantees

To ensure that bridges and similar links needing to talk to a device are available when the device is suspended or resumed, the device hierarchy is walked in a bottom-up order to suspend devices. A top-down order is used to resume those devices.

The ordering of the device hierarchy is defined by the order in which devices get registered: a child can never be registered, probed or resumed before its parent; and can't be removed or suspended after that parent.

The policy is that the device hierarchy should match hardware bus topology. [Or at least the control bus, for devices which use multiple busses.] In particular, this means that a device registration may fail if the parent of the device is suspending (i.e. has been chosen by the PM core as the next device to suspend) or has already suspended, as well as after all of the other devices have been suspended. Device drivers must be prepared to cope with such situations.

System Power Management Phases

Suspending or resuming the system is done in several phases. Different phases are used for suspend-to-idle, shallow (standby), and deep ("suspend-to-RAM") sleep states and the hibernation state ("suspend-to-disk"). Each phase involves executing callbacks for every device before the next phase begins. Not all buses or classes support all these callbacks and not all drivers use all the callbacks. The various phases always run after tasks have been frozen and before they are unfrozen. Furthermore, the `*_noirq` phases run at a time when IRQ handlers have been disabled (except for those marked with the IRQF_NO_SUSPEND flag).

All phases use PM domain, bus, type, class or driver callbacks (that is, methods defined in `dev->pm_domain->ops`, `dev->bus->pm`, `dev->type->pm`, `dev->class->pm` or `dev->driver->pm`). These callbacks are regarded by the PM core as mutually exclusive. Moreover, PM domain callbacks always take precedence over all of the other callbacks and, for example, type callbacks take precedence over bus, class and driver callbacks. To be precise, the following rules are used to determine which callback to execute in the given phase:

1. If `dev->pm_domain` is present, the PM core will choose the callback provided by `dev->pm_domain->ops` for execution.
2. Otherwise, if both `dev->type` and `dev->type->pm` are present, the callback provided by `dev->type->pm` will be chosen for execution.
3. Otherwise, if both `dev->class` and `dev->class->pm` are present, the callback provided by `dev->class->pm` will be chosen for execution.
4. Otherwise, if both `dev->bus` and `dev->bus->pm` are present, the callback provided by `dev->bus->pm` will be chosen for execution.

This allows PM domains and device types to override callbacks provided by bus types or device classes if necessary.

The PM domain, type, class and bus callbacks may in turn invoke device- or driver-specific methods stored in `dev->driver->pm`, but they don't have to do that.

If the subsystem callback chosen for execution is not present, the PM core will execute the corresponding method from the `dev->driver->pm` set instead if there is one.

Entering System Suspend

When the system goes into the freeze, standby or memory sleep state, the phases are: `prepare`, `suspend`, `suspend_late`, `suspend_noirq`.

1. The `prepare` phase is meant to prevent races by preventing new devices from being registered; the PM core would never know that all the children of a device had been suspended if new children could be registered at will. [By contrast, from the PM core's perspective, devices may be unregistered at any time.] Unlike the other suspend-related phases, during the `prepare` phase the device hierarchy is traversed top-down.

   After the `->prepare` callback method returns, no new children may be registered below the device. The method may also prepare the device or driver in some way for the upcoming system power transition, but it should not put the device into a low-power state. Moreover, if the device supports runtime power management, the `->prepare` callback method must not update its state in case it is necessary to resume it from runtime suspend later on.

   For devices supporting runtime power management, the return value of the prepare callback can be used to indicate to the PM core that it may safely leave the device in runtime suspend (if runtime-suspended already), provided that all of the device's descendants are also left in runtime suspend. Namely, if the prepare callback returns a positive number and that happens for all of the descendants of the device too, and all of them (including the device itself) are runtime-suspended, the PM core will skip the `suspend`, `suspend_late` and `suspend_noirq` phases as well as all of the corresponding phases of the subsequent device resume for all of these devices. In that case, the `->complete` callback will be invoked directly after the `->prepare` callback and is entirely responsible for putting the device into a consistent state as appropriate.

   Note that this direct-complete procedure applies even if the device is disabled for runtime PM; only the runtime-PM status matters. It follows that if a device has system-sleep callbacks but does not support runtime PM, then its prepare callback must never return a positive value. This is because all such devices are initially set to runtime-suspended with runtime PM disabled.

   This feature also can be controlled by device drivers by using the `DPM_FLAG_NEVER_SKIP` and `DPM_FLAG_SMART_PREPARE` driver power management flags. [Typically, they are set at the time the driver is probed against the device in question by passing them to the `dev_pm_set_driver_flags()` helper function.] If the first of these flags is set, the PM core will not apply the direct-complete procedure described above to the given device and, consequenty, to any of its ancestors. The second flag, when set, informs the middle layer code (bus types, device types, PM domains, classes) that it should take the return value of the `->prepare` callback provided by the driver into account and it may only return a positive value from its own `->prepare` callback if the driver's one also has returned a positive value.

2. The `->suspend` methods should quiesce the device to stop it from performing I/O. They also may save the device registers and put it into the appropriate low-power state, depending on the bus type the device is on, and they may enable wakeup events.

   However, for devices supporting runtime power management, the `->suspend` methods provided by subsystems (bus types and PM domains in particular) must follow an additional rule regarding what can be done to the devices before their drivers' `->suspend` methods are called. Namely, they can only resume the devices from runtime suspend by calling `pm_runtime_resume()` for them, if that is necessary, and they must not update the state of the devices in any other way at that time (in case the drivers need to resume the devices from runtime suspend in their `->suspend` methods).

3. For a number of devices it is convenient to split suspend into the "quiesce device" and "save device state" phases, in which cases `suspend_late` is meant to do the latter. It is always executed after runtime power management has been disabled for the device in question.

4. The `suspend_noirq` phase occurs after IRQ handlers have been disabled, which means that the driver's interrupt handler will not be called while the callback method is running. The `->suspend_noirq` methods should save the values of the device's registers that weren't saved previously and finally put the device into the appropriate low-power state.

   The majority of subsystems and device drivers need not implement this callback. However, bus types allowing devices to share interrupt vectors, like PCI, generally need it; otherwise a driver might encounter an error during the suspend phase by fielding a shared interrupt generated by some other device after its own device had been set to low power.

At the end of these phases, drivers should have stopped all I/O transactions (DMA, IRQs), saved enough state that they can re-initialize or restore previous state (as needed by the hardware), and placed the device into a low-power state. On many platforms they will gate off one or more clock sources; sometimes they will also switch off power supplies or reduce voltages. [Drivers supporting runtime PM may already have performed some or all of these steps.]

If `device_may_wakeup(dev)()` returns `true`, the device should be prepared for generating hardware wakeup signals to trigger a system wakeup event when the system is in the sleep state. For example, `enable_irq_wake()` might identify GPIO signals hooked up to a switch or other external hardware, and *`pci_enable_wake()`* does something similar for the PCI PME signal.

If any of these callbacks returns an error, the system won't enter the desired low-power state. Instead, the PM core will unwind its actions by resuming all the devices that were suspended.

Leaving System Suspend

When resuming from freeze, standby or memory sleep, the phases are: `resume_noirq`, `resume_early`, `resume`, `complete`.

1. The `->resume_noirq` callback methods should perform any actions needed before the driver's interrupt handlers are invoked. This generally means undoing the actions of the `suspend_noirq` phase. If the bus type permits devices to share interrupt vectors, like PCI, the method should bring the device and its driver into a state in which the driver can recognize if the device is the source of incoming interrupts, if any, and handle them correctly.

   For example, the PCI bus type's `->pm.resume_noirq()` puts the device into the full-power state (D0 in the PCI terminology) and restores the standard configuration registers of the device. Then it calls the device driver's `->pm.resume_noirq()` method to perform device-specific actions.

2. The `->resume_early` methods should prepare devices for the execution of the resume methods. This generally involves undoing the actions of the preceding `suspend_late` phase.

3. The `->resume` methods should bring the device back to its operating state, so that it can perform normal I/O. This generally involves undoing the actions of the `suspend` phase.

4. The `complete` phase should undo the actions of the `prepare` phase. For this reason, unlike the other resume-related phases, during the `complete` phase the device hierarchy is traversed bottom-up.

   Note, however, that new children may be registered below the device as soon as the `->resume` callbacks occur; it's not necessary to wait until the `complete` phase with that.

   Moreover, if the preceding `->prepare` callback returned a positive number, the device may have been left in runtime suspend throughout the whole system suspend and resume (the `suspend`, `suspend_late`, `suspend_noirq` phases of system suspend and the `resume_noirq`, `resume_early`, `resume` phases of system resume may have been skipped for it). In that case, the `->complete` callback is entirely responsible for putting the device into a consistent state after system suspend if necessary. [For example, it may need to queue up a runtime resume request for the device for this purpose.] To check if that is the case, the `->complete` callback can consult the device's `power.direct_complete` flag. Namely, if that flag is set when the `->complete` callback is being run, it has been called directly after the preceding `->prepare` and special actions may be required to make the device work correctly afterward.

At the end of these phases, drivers should be as functional as they were before suspending: I/O can be performed using DMA and IRQs, and the relevant clocks are gated on.

However, the details here may again be platform-specific. For example, some systems support multiple "run" states, and the mode in effect at the end of resume might not be the one which preceded suspension. That means availability of certain clocks or power supplies changed, which could easily affect how a driver works.

Drivers need to be able to handle hardware which has been reset since all of the suspend methods were called, for example by complete reinitialization. This may be the hardest part, and the one most protected by NDA'd documents and chip errata. It's simplest if the hardware state hasn't changed since the suspend was carried out, but that can only be guaranteed if the target system sleep entered was suspend-to-idle. For the other system sleep states that may not be the case (and usually isn't for ACPI-defined system sleep states, like S3).

Drivers must also be prepared to notice that the device has been removed while the system was powered down, whenever that's physically possible. PCMCIA, MMC, USB, Firewire, SCSI, and even IDE are common examples of busses where common Linux platforms will see such removal. Details of how drivers will notice and handle such removals are currently bus-specific, and often involve a separate thread.

These callbacks may return an error value, but the PM core will ignore such errors since there's nothing it can do about them other than printing them in the system log.

Entering Hibernation

Hibernating the system is more complicated than putting it into sleep states, because it involves creating and saving a system image. Therefore there are more phases for hibernation, with a different set of callbacks. These phases always run after tasks have been frozen and enough memory has been freed.

The general procedure for hibernation is to quiesce all devices ("freeze"), create an image of the system memory while everything is stable, reactivate all devices ("thaw"), write the image to permanent storage, and finally shut down the system ("power off"). The phases used to accomplish this are: `prepare`, `freeze`, `freeze_late`, `freeze_noirq`, `thaw_noirq`, `thaw_early`, `thaw`, `complete`, `prepare`, `poweroff`, `poweroff_late`, `poweroff_noirq`.

1. The `prepare` phase is discussed in the "Entering System Suspend" section above.
2. The `->freeze` methods should quiesce the device so that it doesn't generate IRQs or DMA, and they may need to save the values of device registers. However the device does not have to be put in a low-power state, and to save time it's best not to do so. Also, the device should not be prepared to generate wakeup events.
3. The `freeze_late` phase is analogous to the `suspend_late` phase described earlier, except that the device should not be put into a low-power state and should not be allowed to generate wakeup events.
4. The `freeze_noirq` phase is analogous to the `suspend_noirq` phase discussed earlier, except again that the device should not be put into a low-power state and should not be allowed to generate wakeup events.

At this point the system image is created. All devices should be inactive and the contents of memory should remain undisturbed while this happens, so that the image forms an atomic snapshot of the system state.

5. The `thaw_noirq` phase is analogous to the `resume_noirq` phase discussed earlier. The main difference is that its methods can assume the device is in the same state as at the end of the `freeze_noirq` phase.
6. The `thaw_early` phase is analogous to the `resume_early` phase described above. Its methods should undo the actions of the preceding `freeze_late`, if necessary.
7. The `thaw` phase is analogous to the `resume` phase discussed earlier. Its methods should bring the device back to an operating state, so that it can be used for saving the image if necessary.
8. The `complete` phase is discussed in the "Leaving System Suspend" section above.

At this point the system image is saved, and the devices then need to be prepared for the upcoming system shutdown. This is much like suspending them before putting the system into the suspend-to-idle, shallow or deep sleep state, and the phases are similar.

9. The `prepare` phase is discussed above.
10. The `poweroff` phase is analogous to the `suspend` phase.

11. The `poweroff_late` phase is analogous to the `suspend_late` phase.
12. The `poweroff_noirq` phase is analogous to the `suspend_noirq` phase.

The `->poweroff`, `->poweroff_late` and `->poweroff_noirq` callbacks should do essentially the same things as the `->suspend`, `->suspend_late` and `->suspend_noirq` callbacks, respectively. The only notable difference is that they need not store the device register values, because the registers should already have been stored during the `freeze`, `freeze_late` or `freeze_noirq` phases.

Leaving Hibernation

Resuming from hibernation is, again, more complicated than resuming from a sleep state in which the contents of main memory are preserved, because it requires a system image to be loaded into memory and the pre-hibernation memory contents to be restored before control can be passed back to the image kernel.

Although in principle the image might be loaded into memory and the pre-hibernation memory contents restored by the boot loader, in practice this can't be done because boot loaders aren't smart enough and there is no established protocol for passing the necessary information. So instead, the boot loader loads a fresh instance of the kernel, called "the restore kernel", into memory and passes control to it in the usual way. Then the restore kernel reads the system image, restores the pre-hibernation memory contents, and passes control to the image kernel. Thus two different kernel instances are involved in resuming from hibernation. In fact, the restore kernel may be completely different from the image kernel: a different configuration and even a different version. This has important consequences for device drivers and their subsystems.

To be able to load the system image into memory, the restore kernel needs to include at least a subset of device drivers allowing it to access the storage medium containing the image, although it doesn't need to include all of the drivers present in the image kernel. After the image has been loaded, the devices managed by the boot kernel need to be prepared for passing control back to the image kernel. This is very similar to the initial steps involved in creating a system image, and it is accomplished in the same way, using `prepare`, `freeze`, and `freeze_noirq` phases. However, the devices affected by these phases are only those having drivers in the restore kernel; other devices will still be in whatever state the boot loader left them.

Should the restoration of the pre-hibernation memory contents fail, the restore kernel would go through the "thawing" procedure described above, using the `thaw_noirq`, `thaw_early`, `thaw`, and `complete` phases, and then continue running normally. This happens only rarely. Most often the pre-hibernation memory contents are restored successfully and control is passed to the image kernel, which then becomes responsible for bringing the system back to the working state.

To achieve this, the image kernel must restore the devices' pre-hibernation functionality. The operation is much like waking up from a sleep state (with the memory contents preserved), although it involves different phases: `restore_noirq`, `restore_early`, `restore`, `complete`.

1. The `restore_noirq` phase is analogous to the `resume_noirq` phase.
2. The `restore_early` phase is analogous to the `resume_early` phase.
3. The `restore` phase is analogous to the `resume` phase.
4. The `complete` phase is discussed above.

The main difference from `resume[_early|_noirq]` is that `restore[_early|_noirq]` must assume the device has been accessed and reconfigured by the boot loader or the restore kernel. Consequently, the state of the device may be different from the state remembered from the `freeze`, `freeze_late` and `freeze_noirq` phases. The device may even need to be reset and completely re-initialized. In many cases this difference doesn't matter, so the `->resume[_early|_noirq]` and `->restore[_early|_norq]` method pointers can be set to the same routines. Nevertheless, different callback pointers are used in case there is a situation where it actually does matter.

Power Management Notifiers

There are some operations that cannot be carried out by the power management callbacks discussed above, because the callbacks occur too late or too early. To handle these cases, subsystems and device drivers may register power management notifiers that are called before tasks are frozen and after they have been thawed. Generally speaking, the PM notifiers are suitable for performing actions that either require user space to be available, or at least won't interfere with user space.

For details refer to Suspend/Hibernation Notifiers.

Device Low-Power (suspend) States

Device low-power states aren't standard. One device might only handle "on" and "off", while another might support a dozen different versions of "on" (how many engines are active?), plus a state that gets back to "on" faster than from a full "off".

Some buses define rules about what different suspend states mean. PCI gives one example: after the suspend sequence completes, a non-legacy PCI device may not perform DMA or issue IRQs, and any wakeup events it issues would be issued through the PME# bus signal. Plus, there are several PCI-standard device states, some of which are optional.

In contrast, integrated system-on-chip processors often use IRQs as the wakeup event sources (so drivers would call `enable_irq_wake()`) and might be able to treat DMA completion as a wakeup event (sometimes DMA can stay active too, it'd only be the CPU and some peripherals that sleep).

Some details here may be platform-specific. Systems may have devices that can be fully active in certain sleep states, such as an LCD display that's refreshed using DMA while most of the system is sleeping lightly … and its frame buffer might even be updated by a DSP or other non-Linux CPU while the Linux control processor stays idle.

Moreover, the specific actions taken may depend on the target system state. One target system state might allow a given device to be very operational; another might require a hard shut down with re-initialization on resume. And two different target systems might use the same device in different ways; the aforementioned LCD might be active in one product's "standby", but a different product using the same SOC might work differently.

Device Power Management Domains

Sometimes devices share reference clocks or other power resources. In those cases it generally is not possible to put devices into low-power states individually. Instead, a set of devices sharing a power resource can be put into a low-power state together at the same time by turning off the shared power resource. Of course, they also need to be put into the full-power state together, by turning the shared power resource on. A set of devices with this property is often referred to as a power domain. A power domain may also be nested inside another power domain. The nested domain is referred to as the sub-domain of the parent domain.

Support for power domains is provided through the `pm_domain` field of *`struct device`*. This field is a pointer to an object of type *`struct dev_pm_domain`*, defined in `include/linux/pm.h`, providing a set of power management callbacks analogous to the subsystem-level and device driver callbacks that are executed for the given device during all power transitions, instead of the respective subsystem-level callbacks. Specifically, if a device's `pm_domain` pointer is not NULL, the `->suspend()` callback from the object pointed to by it will be executed instead of its subsystem's (e.g. bus type's) `->suspend()` callback and analogously for all of the remaining callbacks. In other words, power management domain callbacks, if defined for the given device, always take precedence over the callbacks provided by the device's subsystem (e.g. bus type).

The support for device power management domains is only relevant to platforms needing to use the same device driver power management callbacks in many different power domain configurations and wanting to avoid incorporating the support for power domains into subsystem-level callbacks, for example by modifying the platform bus type. Other platforms need not implement it or take it into account in any way.

Devices may be defined as IRQ-safe which indicates to the PM core that their runtime PM callbacks may be invoked with disabled interrupts (see `Documentation/power/runtime_pm.txt` for more information). If an IRQ-safe device belongs to a PM domain, the runtime PM of the domain will be disallowed, unless the domain itself is defined as IRQ-safe. However, it makes sense to define a PM domain as IRQ-safe only if all the devices in it are IRQ-safe. Moreover, if an IRQ-safe domain has a parent domain, the runtime PM of the parent is only allowed if the parent itself is IRQ-safe too with the additional restriction that all child domains of an IRQ-safe parent must also be IRQ-safe.

Runtime Power Management

Many devices are able to dynamically power down while the system is still running. This feature is useful for devices that are not being used, and can offer significant power savings on a running system. These devices often support a range of runtime power states, which might use names such as “off”, “sleep”, “idle”, “active”, and so on. Those states will in some cases (like PCI) be partially constrained by the bus the device uses, and will usually include hardware states that are also used in system sleep states.

A system-wide power transition can be started while some devices are in low power states due to runtime power management. The system sleep PM callbacks should recognize such situations and react to them appropriately, but the necessary actions are subsystem-specific.

In some cases the decision may be made at the subsystem level while in other cases the device driver may be left to decide. In some cases it may be desirable to leave a suspended device in that state during a system-wide power transition, but in other cases the device must be put back into the full-power state temporarily, for example so that its system wakeup capability can be disabled. This all depends on the hardware and the design of the subsystem and device driver in question.

If it is necessary to resume a device from runtime suspend during a system-wide transition into a sleep state, that can be done by calling `pm_runtime_resume()` for it from the `->suspend` callback (or its couterpart for transitions related to hibernation) of either the device’s driver or a subsystem responsible for it (for example, a bus type or a PM domain). That is guaranteed to work by the requirement that subsystems must not change the state of devices (possibly except for resuming them from runtime suspend) from their `->prepare` and `->suspend` callbacks (or equivalent) *before* invoking device drivers’ `->suspend` callbacks (or equivalent).

Some bus types and PM domains have a policy to resume all devices from runtime suspend upfront in their `->suspend` callbacks, but that may not be really necessary if the driver of the device can cope with runtime-suspended devices. The driver can indicate that by setting `DPM_FLAG_SMART_SUSPEND` in `power.driver_flags` at the probe time, by passing it to the `dev_pm_set_driver_flags()` helper. That also may cause middle-layer code (bus types, PM domains etc.) to skip the `->suspend_late` and `->suspend_noirq` callbacks provided by the driver if the device remains in runtime suspend at the beginning of the `suspend_late` phase of system-wide suspend (or in the `poweroff_late` phase of hibernation), when runtime PM has been disabled for it, under the assumption that its state should not change after that point until the system-wide transition is over (the PM core itself does that for devices whose “noirq”, “late” and “early” system-wide PM callbacks are executed directly by it). If that happens, the driver’s system-wide resume callbacks, if present, may still be invoked during the subsequent system-wide resume transition and the device’s runtime power management status may be set to “active” before enabling runtime PM for it, so the driver must be prepared to cope with the invocation of its system-wide resume callbacks back-to-back with its `->runtime_suspend` one (without the intervening `->runtime_resume` and so on) and the final state of the device must reflect the “active” runtime PM status in that case.

During system-wide resume from a sleep state it’s easiest to put devices into the full-power state, as explained in `Documentation/power/runtime_pm.txt`. [Refer to that document for more information regarding this particular issue as well as for information on the device runtime power management framework in general.]

However, it often is desirable to leave devices in suspend after system transitions to the working state, especially if those devices had been in runtime suspend before the preceding system-wide suspend (or analogous) transition. Device drivers can use the `DPM_FLAG_LEAVE_SUSPENDED` flag to indicate to the PM core (and middle-layer code) that they prefer the specific devices handled by them to be left suspended and they have no problems with skipping their system-wide resume callbacks for this reason. Whether or not the devices will actually be left in suspend may depend on their state before the given system suspend-resume cycle and on the type of the system transition under way. In particular, devices are not left suspended if that transition is a restore from hibernation, as device states are not guaranteed to be reflected by the information stored in the hibernation image in that case.

The middle-layer code involved in the handling of the device is expected to indicate to the PM core if the device may be left in suspend by setting its `power.may_skip_resume` status bit which is checked by the PM core during the "noirq" phase of the preceding system-wide suspend (or analogous) transition. The middle layer is then responsible for handling the device as appropriate in its "noirq" resume callback, which is executed regardless of whether or not the device is left suspended, but the other resume callbacks (except for `->complete`) will be skipped automatically by the PM core if the device really can be left in suspend.

For devices whose "noirq", "late" and "early" driver callbacks are invoked directly by the PM core, all of the system-wide resume callbacks are skipped if `DPM_FLAG_LEAVE_SUSPENDED` is set and the device is in runtime suspend during the `suspend_noirq` (or analogous) phase or the transition under way is a proper system suspend (rather than anything related to hibernation) and the device's wakeup settings are suitable for runtime PM (that is, it cannot generate wakeup signals at all or it is allowed to wake up the system from sleep).

Suspend/Hibernation Notifiers

```
Copyright (c) 2016 Intel Corp., Rafael J. Wysocki <rafael.j.wysocki@intel.com>
```

There are some operations that subsystems or drivers may want to carry out before hibernation/suspend or after restore/resume, but they require the system to be fully functional, so the drivers' and subsystems' `->suspend()` and `->resume()` or even `->prepare()` and `->complete()` callbacks are not suitable for this purpose.

For example, device drivers may want to upload firmware to their devices after resume/restore, but they cannot do it by calling *`request_firmware()`* from their `->resume()` or `->complete()` callback routines (user land processes are frozen at these points). The solution may be to load the firmware into memory before processes are frozen and upload it from there in the `->resume()` routine. A suspend/hibernation notifier may be used for that.

Subsystems or drivers having such needs can register suspend notifiers that will be called upon the following events by the PM core:

`PM_HIBERNATION_PREPARE` The system is going to hibernate, tasks will be frozen immediately. This is different from PM_SUSPEND_PREPARE below, because in this case additional work is done between the notifiers and the invocation of PM callbacks for the "freeze" transition.

`PM_POST_HIBERNATION` The system memory state has been restored from a hibernation image or an error occurred during hibernation. Device restore callbacks have been executed and tasks have been thawed.

`PM_RESTORE_PREPARE` The system is going to restore a hibernation image. If all goes well, the restored image kernel will issue a PM_POST_HIBERNATION notification.

`PM_POST_RESTORE` An error occurred during restore from hibernation. Device restore callbacks have been executed and tasks have been thawed.

`PM_SUSPEND_PREPARE` The system is preparing for suspend.

`PM_POST_SUSPEND` The system has just resumed or an error occurred during suspend. Device resume callbacks have been executed and tasks have been thawed.

It is generally assumed that whatever the notifiers do for PM_HIBERNATION_PREPARE, should be undone for PM_POST_HIBERNATION. Analogously, operations carried out for PM_SUSPEND_PREPARE should be reversed for PM_POST_SUSPEND.

Moreover, if one of the notifiers fails for the `PM_HIBERNATION_PREPARE` or `PM_SUSPEND_PREPARE` event, the notifiers that have already succeeded for that event will be called for `PM_POST_HIBERNATION` or `PM_POST_SUSPEND`, respectively.

The hibernation and suspend notifiers are called with `pm_mutex` held. They are defined in the usual way, but their last argument is meaningless (it is always NULL).

To register and/or unregister a suspend notifier use `register_pm_notifier()` and `unregister_pm_notifier()`, respectively (both defined in `include/linux/suspend.h`). If you don't need to unregister the notifier, you can also use the `pm_notifier()` macro defined in `include/linux/suspend.h`.

Device Power Management Data Types

struct **dev_pm_ops**
device PM callbacks.

Definition

```
struct dev_pm_ops {
  int (*prepare)(struct device *dev);
  void (*complete)(struct device *dev);
  int (*suspend)(struct device *dev);
  int (*resume)(struct device *dev);
  int (*freeze)(struct device *dev);
  int (*thaw)(struct device *dev);
  int (*poweroff)(struct device *dev);
  int (*restore)(struct device *dev);
  int (*suspend_late)(struct device *dev);
  int (*resume_early)(struct device *dev);
  int (*freeze_late)(struct device *dev);
  int (*thaw_early)(struct device *dev);
  int (*poweroff_late)(struct device *dev);
  int (*restore_early)(struct device *dev);
  int (*suspend_noirq)(struct device *dev);
  int (*resume_noirq)(struct device *dev);
  int (*freeze_noirq)(struct device *dev);
  int (*thaw_noirq)(struct device *dev);
  int (*poweroff_noirq)(struct device *dev);
  int (*restore_noirq)(struct device *dev);
  int (*runtime_suspend)(struct device *dev);
  int (*runtime_resume)(struct device *dev);
  int (*runtime_idle)(struct device *dev);
};
```

Members

`prepare` The principal role of this callback is to prevent new children of the device from being registered after it has returned (the driver's subsystem and generally the rest of the kernel is supposed to prevent new calls to the probe method from being made too once **prepare()** has succeeded). If **prepare()** detects a situation it cannot handle (e.g. registration of a child already in progress), it may return -EAGAIN, so that the PM core can execute it once again (e.g. after a new child has been registered) to recover from the race condition. This method is executed for all kinds of suspend transitions and is followed by one of the suspend callbacks: **suspend()**, **freeze()**, or **poweroff()**. If the transition is a suspend to memory or standby (that is, not related to hibernation), the return value of **prepare()** may be used to indicate to the PM core to leave the device in runtime suspend if applicable. Namely, if **prepare()** returns a positive number, the PM core will understand that as a declaration that the device appears to be runtime-suspended and it may be left in that state during the entire transition and during the subsequent resume if all of its descendants are left in runtime suspend too. If that happens, **complete()** will be executed directly after **prepare()** and it must ensure the proper functioning of the device after the system resume. The PM core executes subsystem-level **prepare()** for all devices before starting to invoke suspend callbacks for any of them, so generally devices may be assumed to be functional or to respond to runtime resume requests while **prepare()** is being executed. However, device drivers may NOT assume anything about the availability of user space at that time and it is NOT valid to request firmware from within **prepare()** (it's too late to do that). It also is NOT valid to allocate substantial amounts of memory from **prepare()** in the GFP_KERNEL mode. [To work around these limitations, drivers may register suspend and hibernation notifiers to be executed before the freezing of tasks.]

`complete` Undo the changes made by **prepare()**. This method is executed for all kinds of resume transitions, following one of the resume callbacks: **resume()**, **thaw()**, **restore()**. Also called if the state transition fails before the driver's suspend callback: **suspend()**, **freeze()** or **poweroff()**, can be executed (e.g. if the suspend callback fails for one of the other devices that the PM core has unsuccessfully attempted to suspend earlier). The PM core executes subsystem-level **complete()** after it has executed the appropriate resume callbacks for all devices. If the corresponding **prepare()** at the beginning of the suspend transition returned a positive number and the device was left in runtime suspend (without executing any suspend and resume callbacks for it), **complete()** will be the only callback executed for the device during resume. In that case, **complete()** must be prepared to do whatever is necessary to ensure the proper functioning of the device after the system resume. To this end, **complete()** can check the power.direct_complete flag of the device to learn whether (unset) or not (set) the previous suspend and resume callbacks have been executed for it.

`suspend` Executed before putting the system into a sleep state in which the contents of main memory are preserved. The exact action to perform depends on the device's subsystem (PM domain, device type, class or bus type), but generally the device must be quiescent after subsystem-level **suspend()** has returned, so that it doesn't do any I/O or DMA. Subsystem-level **suspend()** is executed for all devices after invoking subsystem-level **prepare()** for all of them.

`resume` Executed after waking the system up from a sleep state in which the contents of main memory were preserved. The exact action to perform depends on the device's subsystem, but generally the driver is expected to start working again, responding to hardware events and software requests (the device itself may be left in a low-power state, waiting for a runtime resume to occur). The state of the device at the time its driver's **resume()** callback is run depends on the platform and subsystem the device belongs to. On most platforms, there are no restrictions on availability of resources like clocks during **resume()**. Subsystem-level **resume()** is executed for all devices after invoking subsystem-level **resume_noirq()** for all of them.

`freeze` Hibernation-specific, executed before creating a hibernation image. Analogous to **suspend()**, but it should not enable the device to signal wakeup events or change its power state. The majority of subsystems (with the notable exception of the PCI bus type) expect the driver-level **freeze()** to save the device settings in memory to be used by **restore()** during the subsequent resume from hibernation. Subsystem-level **freeze()** is executed for all devices after invoking subsystem-level **prepare()** for all of them.

`thaw` Hibernation-specific, executed after creating a hibernation image OR if the creation of an image has failed. Also executed after a failing attempt to restore the contents of main memory from such an image. Undo the changes made by the preceding **freeze()**, so the device can be operated in the same way as immediately before the call to **freeze()**. Subsystem-level **thaw()** is executed for all devices after invoking subsystem-level **thaw_noirq()** for all of them. It also may be executed directly after **freeze()** in case of a transition error.

`poweroff` Hibernation-specific, executed after saving a hibernation image. Analogous to **suspend()**, but it need not save the device's settings in memory. Subsystem-level **poweroff()** is executed for all devices after invoking subsystem-level **prepare()** for all of them.

`restore` Hibernation-specific, executed after restoring the contents of main memory from a hibernation image, analogous to **resume()**.

`suspend_late` Continue operations started by **suspend()**. For a number of devices **suspend_late()** may point to the same callback routine as the runtime suspend callback.

`resume_early` Prepare to execute **resume()**. For a number of devices **resume_early()** may point to the same callback routine as the runtime resume callback.

`freeze_late` Continue operations started by **freeze()**. Analogous to **suspend_late()**, but it should not enable the device to signal wakeup events or change its power state.

`thaw_early` Prepare to execute **thaw()**. Undo the changes made by the preceding **freeze_late()**.

`poweroff_late` Continue operations started by **poweroff()**. Analogous to **suspend_late()**, but it need not save the device's settings in memory.

`restore_early` Prepare to execute **restore()**, analogous to **resume_early()**.

`suspend_noirq` Complete the actions started by **suspend()**. Carry out any additional operations required for suspending the device that might be racing with its driver's interrupt handler, which is guaranteed not to run while **suspend_noirq()** is being executed. It generally is expected that the device will be in a low-power state (appropriate for the target system sleep state) after subsystem-level **suspend_noirq()** has returned successfully. If the device can generate system wakeup signals and is enabled to wake up the system, it should be configured to do so at that time. However, depending on the platform and device's subsystem, **suspend()** or **suspend_late()** may be allowed to put the device into the low-power state and configure it to generate wakeup signals, in which case it generally is not necessary to define **suspend_noirq()**.

`resume_noirq` Prepare for the execution of **resume()** by carrying out any operations required for resuming the device that might be racing with its driver's interrupt handler, which is guaranteed not to run while **resume_noirq()** is being executed.

`freeze_noirq` Complete the actions started by **freeze()**. Carry out any additional operations required for freezing the device that might be racing with its driver's interrupt handler, which is guaranteed not to run while **freeze_noirq()** is being executed. The power state of the device should not be changed by either **freeze()**, or **freeze_late()**, or **freeze_noirq()** and it should not be configured to signal system wakeup by any of these callbacks.

`thaw_noirq` Prepare for the execution of **thaw()** by carrying out any operations required for thawing the device that might be racing with its driver's interrupt handler, which is guaranteed not to run while **thaw_noirq()** is being executed.

`poweroff_noirq` Complete the actions started by **poweroff()**. Analogous to **suspend_noirq()**, but it need not save the device's settings in memory.

`restore_noirq` Prepare for the execution of **restore()** by carrying out any operations required for thawing the device that might be racing with its driver's interrupt handler, which is guaranteed not to run while **restore_noirq()** is being executed. Analogous to **resume_noirq()**.

`runtime_suspend` Prepare the device for a condition in which it won't be able to communicate with the CPU(s) and RAM due to power management. This need not mean that the device should be put into a low-power state. For example, if the device is behind a link which is about to be turned off, the device may remain at full power. If the device does go to low power and is capable of generating runtime wakeup events, remote wakeup (i.e., a hardware mechanism allowing the device to request a change of its power state via an interrupt) should be enabled for it.

`runtime_resume` Put the device into the fully active state in response to a wakeup event generated by hardware or at the request of software. If necessary, put the device into the full-power state and restore its registers, so that it is fully operational.

`runtime_idle` Device appears to be inactive and it might be put into a low-power state if all of the necessary conditions are satisfied. Check these conditions, and return 0 if it's appropriate to let the PM core queue a suspend request for the device.

Description

Several device power state transitions are externally visible, affecting the state of pending I/O queues and (for drivers that touch hardware) interrupts, wakeups, DMA, and other hardware state. There may also be internal transitions to various low-power modes which are transparent to the rest of the driver stack (such as a driver that's ON gating off clocks which are not in active use).

The externally visible transitions are handled with the help of callbacks included in this structure in such a way that, typically, two levels of callbacks are involved. First, the PM core executes callbacks provided by PM domains, device types, classes and bus types. They are the subsystem-level callbacks expected to execute callbacks provided by device drivers, although they may choose not to do that. If the driver callbacks are executed, they have to collaborate with the subsystem-level callbacks to achieve the goals appropriate for the given system transition, given transition phase and the subsystem the device belongs to.

All of the above callbacks, except for **complete()**, return error codes. However, the error codes returned by **resume()**, **thaw()**, **restore()**, **resume_noirq()**, **thaw_noirq()**, and **restore_noirq()**, do not cause the PM core to abort the resume transition during which they are returned. The error codes returned in those cases are only printed to the system logs for debugging purposes. Still, it is recommended that drivers only return error codes from their resume methods in case of an unrecoverable failure (i.e. when the device being handled refuses to resume and becomes unusable) to allow the PM core to be modified in the future, so that it can avoid attempting to handle devices that failed to resume and their children.

It is allowed to unregister devices while the above callbacks are being executed. However, a callback routine MUST NOT try to unregister the device it was called for, although it may unregister children of that device (for example, if it detects that a child was unplugged while the system was asleep).

There also are callbacks related to runtime power management of devices. Again, as a rule these callbacks are executed by the PM core for subsystems (PM domains, device types, classes and bus types) and the subsystem-level callbacks are expected to invoke the driver callbacks. Moreover, the exact actions to be performed by a device driver's callbacks generally depend on the platform and subsystem the device belongs to.

Refer to Documentation/power/runtime_pm.txt for more information about the role of the **runtime_suspend()**, **runtime_resume()** and **runtime_idle()** callbacks in device runtime power management.

struct **dev_pm_domain**
power management domain representation.

Definition

```
struct dev_pm_domain {
  struct dev_pm_ops       ops;
  void (*detach)(struct device *dev, bool power_off);
  int (*activate)(struct device *dev);
  void (*sync)(struct device *dev);
  void (*dismiss)(struct device *dev);
};
```

Members

ops Power management operations associated with this domain.

detach Called when removing a device from the domain.

activate Called before executing probe routines for bus types and drivers.

sync Called after successful driver probe.

dismiss Called after unsuccessful driver probe and after driver removal.

Description

Power domains provide callbacks that are executed during system suspend, hibernation, system resume and during runtime PM transitions instead of subsystem-level and driver-level callbacks.

CHAPTER

FOUR

BUS-INDEPENDENT DEVICE ACCESSES

Author Matthew Wilcox

Author Alan Cox

Introduction

Linux provides an API which abstracts performing IO across all busses and devices, allowing device drivers to be written independently of bus type.

Memory Mapped IO

Getting Access to the Device

The most widely supported form of IO is memory mapped IO. That is, a part of the CPU's address space is interpreted not as accesses to memory, but as accesses to a device. Some architectures define devices to be at a fixed address, but most have some method of discovering devices. The PCI bus walk is a good example of such a scheme. This document does not cover how to receive such an address, but assumes you are starting with one. Physical addresses are of type unsigned long.

This address should not be used directly. Instead, to get an address suitable for passing to the accessor functions described below, you should call *`ioremap()`*. An address suitable for accessing the device will be returned to you.

After you've finished using the device (say, in your module's exit routine), call `iounmap()` in order to return the address space to the kernel. Most architectures allocate new address space each time you call *`ioremap()`*, and they can run out unless you call `iounmap()`.

Accessing the device

The part of the interface most used by drivers is reading and writing memory-mapped registers on the device. Linux provides interfaces to read and write 8-bit, 16-bit, 32-bit and 64-bit quantities. Due to a historical accident, these are named byte, word, long and quad accesses. Both read and write accesses are supported; there is no prefetch support at this time.

The functions are named readb(), readw(), readl(), readq(), readb_relaxed(), readw_relaxed(), readl_relaxed(), readq_relaxed(), writeb(), writew(), writel() and writeq().

Some devices (such as framebuffers) would like to use larger transfers than 8 bytes at a time. For these devices, the `memcpy_toio()`, `memcpy_fromio()` and `memset_io()` functions are provided. Do not use memset or memcpy on IO addresses; they are not guaranteed to copy data in order.

The read and write functions are defined to be ordered. That is the compiler is not permitted to reorder the I/O sequence. When the ordering can be compiler optimised, you can use __readb() and friends to indicate the relaxed ordering. Use this with care.

While the basic functions are defined to be synchronous with respect to each other and ordered with respect to each other the busses the devices sit on may themselves have asynchronicity. In particular many authors are burned by the fact that PCI bus writes are posted asynchronously. A driver author must issue a read from the same device to ensure that writes have occurred in the specific cases the author cares. This kind of property cannot be hidden from driver writers in the API. In some cases, the read used to flush the device may be expected to fail (if the card is resetting, for example). In that case, the read should be done from config space, which is guaranteed to soft-fail if the card doesn't respond.

The following is an example of flushing a write to a device when the driver would like to ensure the write's effects are visible prior to continuing execution:

```
static inline void
qla1280_disable_intrs(struct scsi_qla_host *ha)
{
    struct device_reg *reg;

    reg = ha->iobase;
    /* disable risc and host interrupts */
    WRT_REG_WORD(&reg->ictrl, 0);
    /*
     * The following read will ensure that the above write
     * has been received by the device before we return from this
     * function.
     */
    RD_REG_WORD(&reg->ictrl);
    ha->flags.ints_enabled = 0;
}
```

In addition to write posting, on some large multiprocessing systems (e.g. SGI Challenge, Origin and Altix machines) posted writes won't be strongly ordered coming from different CPUs. Thus it's important to properly protect parts of your driver that do memory-mapped writes with locks and use the `mmiowb()` to make sure they arrive in the order intended. Issuing a regular readX() will also ensure write ordering, but should only be used when the driver has to be sure that the write has actually arrived at the device (not that it's simply ordered with respect to other writes), since a full readX() is a relatively expensive operation.

Generally, one should use `mmiowb()` prior to releasing a spinlock that protects regions using `writeb()` or similar functions that aren't surrounded by readb() calls, which will ensure ordering and flushing. The following pseudocode illustrates what might occur if write ordering isn't guaranteed via `mmiowb()` or one of the readX() functions:

```
CPU A:  spin_lock_irqsave(&dev_lock, flags)
CPU A:  ...
CPU A:  writel(newval, ring_ptr);
CPU A:  spin_unlock_irqrestore(&dev_lock, flags)
        ...
CPU B:  spin_lock_irqsave(&dev_lock, flags)
CPU B:  writel(newval2, ring_ptr);
CPU B:  ...
CPU B:  spin_unlock_irqrestore(&dev_lock, flags)
```

In the case above, newval2 could be written to ring_ptr before newval. Fixing it is easy though:

```
CPU A:  spin_lock_irqsave(&dev_lock, flags)
CPU A:  ...
CPU A:  writel(newval, ring_ptr);
CPU A:  mmiowb(); /* ensure no other writes beat us to the device */
CPU A:  spin_unlock_irqrestore(&dev_lock, flags)
        ...
CPU B:  spin_lock_irqsave(&dev_lock, flags)
CPU B:  writel(newval2, ring_ptr);
CPU B:  ...
CPU B:  mmiowb();
CPU B:  spin_unlock_irqrestore(&dev_lock, flags)
```

See tg3.c for a real world example of how to use `mmiowb()`

PCI ordering rules also guarantee that PIO read responses arrive after any outstanding DMA writes from that bus, since for some devices the result of a readb() call may signal to the driver that a DMA transaction is complete. In many cases, however, the driver may want to indicate that the next readb() call has no relation to any previous DMA writes performed by the device. The driver can use readb_relaxed() for these cases, although only some platforms will honor the relaxed semantics. Using the relaxed read functions will provide significant performance benefits on platforms that support it. The qla2xxx driver provides examples of how to use readX_relaxed(). In many cases, a majority of the driver's readX() calls can safely be converted to readX_relaxed() calls, since only a few will indicate or depend on DMA completion.

Port Space Accesses

Port Space Explained

Another form of IO commonly supported is Port Space. This is a range of addresses separate to the normal memory address space. Access to these addresses is generally not as fast as accesses to the memory mapped addresses, and it also has a potentially smaller address space.

Unlike memory mapped IO, no preparation is required to access port space.

Accessing Port Space

Accesses to this space are provided through a set of functions which allow 8-bit, 16-bit and 32-bit accesses; also known as byte, word and long. These functions are `inb()`, `inw()`, `inl()`, `outb()`, `outw()` and `outl()`.

Some variants are provided for these functions. Some devices require that accesses to their ports are slowed down. This functionality is provided by appending a _p to the end of the function. There are also equivalents to memcpy. The `ins()` and `outs()` functions copy bytes, words or longs to the given port.

Public Functions Provided

phys_addr_t **`virt_to_phys`**(volatile void * *address*)
map virtual addresses to physical

Parameters

`volatile void * address` address to remap

Description

The returned physical address is the physical (CPU) mapping for the memory address given. It is only valid to use this function on addresses directly mapped or allocated via kmalloc.

This function does not give bus mappings for DMA transfers. In almost all conceivable cases a device driver should not be using this function

void * **`phys_to_virt`**(phys_addr_t *address*)
map physical address to virtual

Parameters

`phys_addr_t address` address to remap

Description

The returned virtual address is a current CPU mapping for the memory address given. It is only valid to use this function on addresses that have a kernel mapping

This function does not handle bus mappings for DMA transfers. In almost all conceivable cases a device driver should not be using this function

void __iomem * **ioremap**(resource_size_t *offset*, unsigned long *size*)
map bus memory into CPU space

Parameters

resource_size_t offset bus address of the memory

unsigned long size size of the resource to map

Description

ioremap performs a platform specific sequence of operations to make bus memory CPU accessible via the readb/readw/readl/writeb/ writew/writel functions and the other mmio helpers. The returned address is not guaranteed to be usable directly as a virtual address.

If the area you are trying to map is a PCI BAR you should have a look at `pci_iomap()`.

void __iomem * **pci_iomap_range**(struct pci_dev * *dev*, int *bar*, unsigned long *offset*, unsigned long *maxlen*)
create a virtual mapping cookie for a PCI BAR

Parameters

struct pci_dev * dev PCI device that owns the BAR

int bar BAR number

unsigned long offset map memory at the given offset in BAR

unsigned long maxlen max length of the memory to map

Description

Using this function you will get a __iomem address to your device BAR. You can access it using ioread*() and iowrite*(). These functions hide the details if this is a MMIO or PIO address space and will just do what you expect from them in the correct way.

maxlen specifies the maximum length to map. If you want to get access to the complete BAR from offset to the end, pass `0` here.

void __iomem * **pci_iomap_wc_range**(struct pci_dev * *dev*, int *bar*, unsigned long *offset*, unsigned long *maxlen*)
create a virtual WC mapping cookie for a PCI BAR

Parameters

struct pci_dev * dev PCI device that owns the BAR

int bar BAR number

unsigned long offset map memory at the given offset in BAR

unsigned long maxlen max length of the memory to map

Description

Using this function you will get a __iomem address to your device BAR. You can access it using ioread*() and iowrite*(). These functions hide the details if this is a MMIO or PIO address space and will just do what you expect from them in the correct way. When possible write combining is used.

maxlen specifies the maximum length to map. If you want to get access to the complete BAR from offset to the end, pass `0` here.

void __iomem * **pci_iomap**(struct pci_dev * *dev*, int *bar*, unsigned long *maxlen*)
create a virtual mapping cookie for a PCI BAR

Parameters

`struct pci_dev * dev` PCI device that owns the BAR

`int bar` BAR number

`unsigned long maxlen` length of the memory to map

Description

Using this function you will get a __iomem address to your device BAR. You can access it using ioread*() and iowrite*(). These functions hide the details if this is a MMIO or PIO address space and will just do what you expect from them in the correct way.

maxlen specifies the maximum length to map. If you want to get access to the complete BAR without checking for its length first, pass `0` here.

void __iomem * **`pci_iomap_wc`**(struct pci_dev * *dev*, int *bar*, unsigned long *maxlen*)
create a virtual WC mapping cookie for a PCI BAR

Parameters

`struct pci_dev * dev` PCI device that owns the BAR

`int bar` BAR number

`unsigned long maxlen` length of the memory to map

Description

Using this function you will get a __iomem address to your device BAR. You can access it using ioread*() and iowrite*(). These functions hide the details if this is a MMIO or PIO address space and will just do what you expect from them in the correct way. When possible write combining is used.

maxlen specifies the maximum length to map. If you want to get access to the complete BAR without checking for its length first, pass `0` here.

CHAPTER

FIVE

BUFFER SHARING AND SYNCHRONIZATION

The dma-buf subsystem provides the framework for sharing buffers for hardware (DMA) access across multiple device drivers and subsystems, and for synchronizing asynchronous hardware access.

This is used, for example, by drm "prime" multi-GPU support, but is of course not limited to GPU use cases.

The three main components of this are: (1) dma-buf, representing a sg_table and exposed to userspace as a file descriptor to allow passing between devices, (2) fence, which provides a mechanism to signal when one device as finished access, and (3) reservation, which manages the shared or exclusive fence(s) associated with the buffer.

Shared DMA Buffers

This document serves as a guide to device-driver writers on what is the dma-buf buffer sharing API, how to use it for exporting and using shared buffers.

Any device driver which wishes to be a part of DMA buffer sharing, can do so as either the 'exporter' of buffers, or the 'user' or 'importer' of buffers.

Say a driver A wants to use buffers created by driver B, then we call B as the exporter, and A as buffer-user/importer.

The exporter

- implements and manages operations in *`struct dma_buf_ops`* for the buffer,
- allows other users to share the buffer by using dma_buf sharing APIs,
- manages the details of buffer allocation, wrapped int a *`struct dma_buf`*,
- decides about the actual backing storage where this allocation happens,
- and takes care of any migration of scatterlist - for all (shared) users of this buffer.

The buffer-user

- is one of (many) sharing users of the buffer.
- doesn't need to worry about how the buffer is allocated, or where.
- and needs a mechanism to get access to the scatterlist that makes up this buffer in memory, mapped into its own address space, so it can access the same area of memory. This interface is provided by *`struct dma_buf_attachment`*.

Any exporters or users of the dma-buf buffer sharing framework must have a 'select DMA_SHARED_BUFFER' in their respective Kconfigs.

Userspace Interface Notes

Mostly a DMA buffer file descriptor is simply an opaque object for userspace, and hence the generic interface exposed is very minimal. There's a few things to consider though:

- Since kernel 3.12 the dma-buf FD supports the llseek system call, but only with offset=0 and whence=SEEK_END|SEEK_SET. SEEK_SET is supported to allow the usual size discover pattern size = SEEK_END(0); SEEK_SET(0). Every other llseek operation will report -EINVAL.

  If llseek on dma-buf FDs isn't support the kernel will report -ESPIPE for all cases. Userspace can use this to detect support for discovering the dma-buf size using llseek.

- In order to avoid fd leaks on exec, the FD_CLOEXEC flag must be set on the file descriptor. This is not just a resource leak, but a potential security hole. It could give the newly exec'd application access to buffers, via the leaked fd, to which it should otherwise not be permitted access.

  The problem with doing this via a separate fcntl() call, versus doing it atomically when the fd is created, is that this is inherently racy in a multi-threaded app[3]. The issue is made worse when it is library code opening/creating the file descriptor, as the application may not even be aware of the fd's.

  To avoid this problem, userspace must have a way to request O_CLOEXEC flag be set when the dma-buf fd is created. So any API provided by the exporting driver to create a dmabuf fd must provide a way to let userspace control setting of O_CLOEXEC flag passed in to dma_buf_fd().

- Memory mapping the contents of the DMA buffer is also supported. See the discussion below on *CPU Access to DMA Buffer Objects* for the full details.

- The DMA buffer FD is also pollable, see *Fence Poll Support* below for details.

Basic Operation and Device DMA Access

For device DMA access to a shared DMA buffer the usual sequence of operations is fairly simple:

1. The exporter defines his exporter instance using *`DEFINE_DMA_BUF_EXPORT_INFO()`* and calls *`dma_buf_export()`* to wrap a private buffer object into a *`dma_buf`*. It then exports that *`dma_buf`* to userspace as a file descriptor by calling *`dma_buf_fd()`*.

2. Userspace passes this file-descriptors to all drivers it wants this buffer to share with: First the filedescriptor is converted to a *`dma_buf`* using *`dma_buf_get()`*. Then the buffer is attached to the device using *`dma_buf_attach()`*.

   Up to this stage the exporter is still free to migrate or reallocate the backing storage.

3. Once the buffer is attached to all devices userspace can initiate DMA access to the shared buffer. In the kernel this is done by calling *`dma_buf_map_attachment()`* and *`dma_buf_unmap_attachment()`*.

4. Once a driver is done with a shared buffer it needs to call *`dma_buf_detach()`* (after cleaning up any mappings) and then release the reference acquired with dma_buf_get by calling *`dma_buf_put()`*.

For the detailed semantics exporters are expected to implement see *`dma_buf_ops`*.

CPU Access to DMA Buffer Objects

There are mutliple reasons for supporting CPU access to a dma buffer object:

- Fallback operations in the kernel, for example when a device is connected over USB and the kernel needs to shuffle the data around first before sending it away. Cache coherency is handled by braketing any transactions with calls to *`dma_buf_begin_cpu_access()`* and *`dma_buf_end_cpu_access()`* access.

  To support dma_buf objects residing in highmem cpu access is page-based using an api similar to kmap. Accessing a dma_buf is done in aligned chunks of PAGE_SIZE size. Before accessing a chunk it needs to be mapped, which returns a pointer in kernel virtual address space. Afterwards the chunk needs to be unmapped again. There is no limit on how often a given chunk can be mapped and unmapped, i.e. the importer does not need to call begin_cpu_access again before mapping the same chunk again.

Interfaces:: void *dma_buf_kmap(struct dma_buf *, unsigned long); void dma_buf_kunmap(struct dma_buf *, unsigned long, void *);

There are also atomic variants of these interfaces. Like for kmap they facilitate non-blocking fast-paths. Neither the importer nor the exporter (in the callback) is allowed to block when using these.

Interfaces:: void *dma_buf_kmap_atomic(struct dma_buf *, unsigned long); void dma_buf_kunmap_atomic(struct dma_buf *, unsigned long, void *);

For importers all the restrictions of using kmap apply, like the limited supply of kmap_atomic slots. Hence an importer shall only hold onto at max 2 atomic dma_buf kmaps at the same time (in any given process context).

dma_buf kmap calls outside of the range specified in begin_cpu_access are undefined. If the range is not PAGE_SIZE aligned, kmap needs to succeed on the partial chunks at the beginning and end but may return stale or bogus data outside of the range (in these partial chunks).

Note that these calls need to always succeed. The exporter needs to complete any preparations that might fail in begin_cpu_access.

For some cases the overhead of kmap can be too high, a vmap interface is introduced. This interface should be used very carefully, as vmalloc space is a limited resources on many architectures.

Interfaces:: void *dma_buf_vmap(struct dma_buf *dmabuf) void dma_buf_vunmap(struct dma_buf *dmabuf, void *vaddr)

The vmap call can fail if there is no vmap support in the exporter, or if it runs out of vmalloc space. Fallback to kmap should be implemented. Note that the dma-buf layer keeps a reference count for all vmap access and calls down into the exporter's vmap function only when no vmapping exists, and only unmaps it once. Protection against concurrent vmap/vunmap calls is provided by taking the dma_buf->lock mutex.

- For full compatibility on the importer side with existing userspace interfaces, which might already support mmap'ing buffers. This is needed in many processing pipelines (e.g. feeding a software rendered image into a hardware pipeline, thumbnail creation, snapshots, ...). Also, Android's ION framework already supported this and for DMA buffer file descriptors to replace ION buffers mmap support was needed.

  There is no special interfaces, userspace simply calls mmap on the dma-buf fd. But like for CPU access there's a need to braket the actual access, which is handled by the ioctl (DMA_BUF_IOCTL_SYNC). Note that DMA_BUF_IOCTL_SYNC can fail with -EAGAIN or -EINTR, in which case it must be restarted.

  Some systems might need some sort of cache coherency management e.g. when CPU and GPU domains are being accessed through dma-buf at the same time. To circumvent this problem there are begin/end coherency markers, that forward directly to existing dma-buf device drivers vfunc hooks. Userspace can make use of those markers through the DMA_BUF_IOCTL_SYNC ioctl. The sequence would be used like following:

    - mmap dma-buf fd
    - for each drawing/upload cycle in CPU 1. SYNC_START ioctl, 2. read/write to mmap area 3. SYNC_END ioctl. This can be repeated as often as you want (with the new data being consumed by say the GPU or the scanout device)
    - munmap once you don't need the buffer any more

    For correctness and optimal performance, it is always required to use SYNC_START and SYNC_END before and after, respectively, when accessing the mapped address. Userspace cannot rely on coherent access, even when there are systems where it just works without calling these ioctls.

- And as a CPU fallback in userspace processing pipelines.

  Similar to the motivation for kernel cpu access it is again important that the userspace code of a given importing subsystem can use the same interfaces with a imported dma-buf buffer object as with a native buffer object. This is especially important for drm where the userspace part of contemporary

OpenGL, X, and other drivers is huge, and reworking them to use a different way to mmap a buffer rather invasive.

The assumption in the current dma-buf interfaces is that redirecting the initial mmap is all that's needed. A survey of some of the existing subsystems shows that no driver seems to do any nefarious thing like syncing up with outstanding asynchronous processing on the device or allocating special resources at fault time. So hopefully this is good enough, since adding interfaces to intercept pagefaults and allow pte shootdowns would increase the complexity quite a bit.

Interface::

int dma_buf_mmap(struct dma_buf *, struct vm_area_struct *, unsigned long);

If the importing subsystem simply provides a special-purpose mmap call to set up a mapping in userspace, calling do_mmap with dma_buf->file will equally achieve that for a dma-buf object.

Fence Poll Support

To support cross-device and cross-driver synchronization of buffer access implicit fences (represented internally in the kernel with `struct fence`) can be attached to a *`dma_buf`*. The glue for that and a few related things are provided in the *`reservation_object`* structure.

Userspace can query the state of these implicitly tracked fences using `poll()` and related system calls:

- Checking for EPOLLIN, i.e. read access, can be use to query the state of the most recent write or exclusive fence.
- Checking for EPOLLOUT, i.e. write access, can be used to query the state of all attached fences, shared and exclusive ones.

Note that this only signals the completion of the respective fences, i.e. the DMA transfers are complete. Cache flushing and any other necessary preparations before CPU access can begin still need to happen.

Kernel Functions and Structures Reference

struct *dma_buf* * **`dma_buf_export`**(const struct *dma_buf_export_info* * *exp_info*)
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.

Parameters

`const struct dma_buf_export_info * 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.

For most cases the easiest way to create **exp_info** is through the `DEFINE_DMA_BUF_EXPORT_INFO` macro.

int **`dma_buf_fd`**(struct *dma_buf* * *dmabuf*, int *flags*)
returns a file descriptor for the given dma_buf

Parameters

`struct dma_buf * dmabuf` [in] pointer to dma_buf for which fd is required.

`int flags` [in] flags to give to fd

Description

On success, returns an associated 'fd'. Else, returns error.

struct *dma_buf* * **dma_buf_get**(int *fd*)
returns the dma_buf structure related to an fd

Parameters

int 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.

void **dma_buf_put**(struct *dma_buf* * *dmabuf*)
decreases refcount of the buffer

Parameters

struct dma_buf * dmabuf [in] buffer to reduce refcount of

Description

Uses file's refcounting done implicitly by `fput()`.

If, as a result of this call, the refcount becomes 0, the 'release' file operation related to this fd is called. It calls *dma_buf_ops.release* vfunc in turn, and frees the memory allocated for dmabuf when exported.

struct *dma_buf_attachment* * **dma_buf_attach**(struct *dma_buf* * *dmabuf*, struct *device* * *dev*)
Add the device to dma_buf's attachments list; optionally, calls `attach()` of dma_buf_ops to allow device-specific attach functionality

Parameters

struct dma_buf * dmabuf [in] buffer to attach device to.

struct device * dev [in] device to be attached.

Description

Returns struct dma_buf_attachment pointer for this attachment. Attachments must be cleaned up by calling *dma_buf_detach()*.

Return

A pointer to newly created *dma_buf_attachment* on success, or a negative error code wrapped into a pointer on failure.

Note that this can fail if the backing storage of **dmabuf** is in a place not accessible to **dev**, and cannot be moved to a more suitable place. This is indicated with the error code -EBUSY.

void **dma_buf_detach**(struct *dma_buf* * *dmabuf*, struct *dma_buf_attachment* * *attach*)
Remove the given attachment from dmabuf's attachments list; optionally calls `detach()` of dma_buf_ops for device-specific detach

Parameters

struct dma_buf * dmabuf [in] buffer to detach from.

struct dma_buf_attachment * attach [in] attachment to be detached; is free'd after this call.

Description

Clean up a device attachment obtained by calling *dma_buf_attach()*.

struct sg_table * **dma_buf_map_attachment**(struct *dma_buf_attachment* * *attach*, enum dma_data_direction *direction*)
Returns the scatterlist table of the attachment; mapped into _device_ address space. Is a wrapper for `map_dma_buf()` of the dma_buf_ops.

Parameters

struct dma_buf_attachment * attach [in] attachment whose scatterlist is to be returned

enum dma_data_direction direction [in] direction of DMA transfer

Description

Returns sg_table containing the scatterlist to be returned; returns ERR_PTR on error. May return -EINTR if it is interrupted by a signal.

A mapping must be unmapped by using *dma_buf_unmap_attachment()*. Note that the underlying backing storage is pinned for as long as a mapping exists, therefore users/importers should not hold onto a mapping for undue amounts of time.

void **dma_buf_unmap_attachment**(struct *dma_buf_attachment* * *attach*, struct sg_table * *sg_table*, enum dma_data_direction *direction*)

unmaps and decreases usecount of the buffer;might deallocate the scatterlist associated. Is a wrapper for `unmap_dma_buf()` of dma_buf_ops.

Parameters

struct dma_buf_attachment * attach [in] attachment to unmap buffer from

struct sg_table * sg_table [in] scatterlist info of the buffer to unmap

enum dma_data_direction direction [in] direction of DMA transfer

Description

This unmaps a DMA mapping for **attached** obtained by *dma_buf_map_attachment()*.

int **dma_buf_begin_cpu_access**(struct *dma_buf* * *dmabuf*, enum dma_data_direction *direction*)

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.

Parameters

struct dma_buf * dmabuf [in] buffer to prepare cpu access for.

enum dma_data_direction direction [in] length of range for cpu access.

Description

After the cpu access is complete the caller should call *dma_buf_end_cpu_access()*. Only when cpu access is braketed by both calls is it guaranteed to be coherent with other DMA access.

Can return negative error values, returns 0 on success.

int **dma_buf_end_cpu_access**(struct *dma_buf* * *dmabuf*, enum dma_data_direction *direction*)

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.

Parameters

struct dma_buf * dmabuf [in] buffer to complete cpu access for.

enum dma_data_direction direction [in] length of range for cpu access.

Description

This terminates CPU access started with *dma_buf_begin_cpu_access()*.

Can return negative error values, returns 0 on success.

void * **dma_buf_kmap_atomic**(struct *dma_buf* * *dmabuf*, unsigned long *page_num*)

Map a page of the buffer object into kernel address space. The same restrictions as for kmap_atomic and friends apply.

Parameters

struct dma_buf * dmabuf [in] buffer to map page from.

unsigned long 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.

void **dma_buf_kunmap_atomic**(struct *dma_buf* * *dmabuf*, unsigned long *page_num*, void * *vaddr*)
Unmap a page obtained by dma_buf_kmap_atomic.

Parameters

struct dma_buf * dmabuf [in] buffer to unmap page from.

unsigned long page_num [in] page in PAGE_SIZE units to unmap.

void * vaddr [in] kernel space pointer obtained from dma_buf_kmap_atomic.

Description

This call must always succeed.

void * **dma_buf_kmap**(struct *dma_buf* * *dmabuf*, unsigned long *page_num*)
Map a page of the buffer object into kernel address space. The same restrictions as for kmap and friends apply.

Parameters

struct dma_buf * dmabuf [in] buffer to map page from.

unsigned long 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.

void **dma_buf_kunmap**(struct *dma_buf* * *dmabuf*, unsigned long *page_num*, void * *vaddr*)
Unmap a page obtained by dma_buf_kmap.

Parameters

struct dma_buf * dmabuf [in] buffer to unmap page from.

unsigned long page_num [in] page in PAGE_SIZE units to unmap.

void * vaddr [in] kernel space pointer obtained from dma_buf_kmap.

Description

This call must always succeed.

int **dma_buf_mmap**(struct *dma_buf* * *dmabuf*, struct vm_area_struct * *vma*, unsigned long *pgoff*)
Setup up a userspace mmap with the given vma

Parameters

struct dma_buf * dmabuf [in] buffer that should back the vma

struct vm_area_struct * vma [in] vma for the mmap

unsigned long 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.

void * **dma_buf_vmap**(struct *dma_buf* * *dmabuf*)
Create virtual mapping for the buffer object into kernel address space. Same restrictions as for vmap and friends apply.

Parameters

struct dma_buf * 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.

void **dma_buf_vunmap**(struct *dma_buf* * *dmabuf*, void * *vaddr*)
Unmap a vmap obtained by dma_buf_vmap.

Parameters

struct dma_buf * dmabuf [in] buffer to vunmap

void * vaddr [in] vmap to vunmap

struct **dma_buf_ops**
operations possible on struct dma_buf

Definition

```
struct dma_buf_ops {
  int (*attach)(struct dma_buf *, struct device *, struct dma_buf_attachment *);
  void (*detach)(struct dma_buf *, struct dma_buf_attachment *);
  struct sg_table * (*map_dma_buf)(struct dma_buf_attachment *, enum dma_data_direction);
  void (*unmap_dma_buf)(struct dma_buf_attachment *,struct sg_table *, enum dma_data_direction);
  void (*release)(struct dma_buf *);
  int (*begin_cpu_access)(struct dma_buf *, enum dma_data_direction);
  int (*end_cpu_access)(struct dma_buf *, enum dma_data_direction);
  void *(*map_atomic)(struct dma_buf *, unsigned long);
  void (*unmap_atomic)(struct dma_buf *, unsigned long, void *);
  void *(*map)(struct dma_buf *, unsigned long);
  void (*unmap)(struct dma_buf *, unsigned long, void *);
  int (*mmap)(struct dma_buf *, struct vm_area_struct *vma);
  void *(*vmap)(struct dma_buf *);
  void (*vunmap)(struct dma_buf *, void *vaddr);
};
```

Members

attach This is called from *dma_buf_attach()* to make sure that a given *device* can access the provided *dma_buf*. Exporters which support buffer objects in special locations like VRAM or device-specific carveout areas should check whether the buffer could be move to system memory (or directly accessed by the provided device), and otherwise need to fail the attach operation.

The exporter should also in general check whether the current allocation fullfills the DMA constraints of the new device. If this is not the case, and the allocation cannot be moved, it should also fail the attach operation.

Any exporter-private housekeeping data can be stored in the *dma_buf_attachment.priv* pointer.

This callback is optional.

Returns:

0 on success, negative error code on failure. It might return -EBUSY to signal that backing storage is already allocated and incompatible with the requirements of requesting device.

detach This is called by *dma_buf_detach()* to release a *dma_buf_attachment*. Provided so that exporters can clean up any housekeeping for an *dma_buf_attachment*.

This callback is optional.

map_dma_buf This is called by *dma_buf_map_attachment()* and is used to map a shared *dma_buf* into device address space, and it is mandatory. It can only be called if **attach** has been called successfully. This essentially pins the DMA buffer into place, and it cannot be moved any more

This call may sleep, e.g. when the backing storage first needs to be allocated, or moved to a location suitable for all currently attached devices.

Note that any specific buffer attributes required for this function should get added to device_dma_parameters accessible via *device.dma_params* from the *dma_buf_attachment*. The **attach** callback should also check these constraints.

If this is being called for the first time, the exporter can now choose to scan through the list of attachments for this buffer, collate the requirements of the attached devices, and choose an appropriate backing storage for the buffer.

Based on enum dma_data_direction, it might be possible to have multiple users accessing at the same time (for reading, maybe), or any other kind of sharing that the exporter might wish to make available to buffer-users.

Returns:

A `sg_table` scatter list of or the backing storage of the DMA buffer, already mapped into the device address space of the *device* attached with the provided *dma_buf_attachment*.

On failure, returns a negative error value wrapped into a pointer. May also return -EINTR when a signal was received while being blocked.

unmap_dma_buf This is called by *dma_buf_unmap_attachment()* and should unmap and release the `sg_table` allocated in **map_dma_buf**, and it is mandatory. It should also unpin the backing storage if this is the last mapping of the DMA buffer, it the exporter supports backing storage migration.

release Called after the last dma_buf_put to release the *dma_buf*, and mandatory.

begin_cpu_access This is called from *dma_buf_begin_cpu_access()* and allows the exporter to ensure that the memory is actually available for cpu access - the exporter might need to allocate or swap-in and pin the backing storage. The exporter also needs to ensure that cpu access is coherent for the access direction. The direction can be used by the exporter to optimize the cache flushing, i.e. access with a different direction (read instead of write) might return stale or even bogus data (e.g. when the exporter needs to copy the data to temporary storage).

This callback is optional.

FIXME: This is both called through the DMA_BUF_IOCTL_SYNC command from userspace (where storage shouldn't be pinned to avoid handing de-factor mlock rights to userspace) and for the kernel-internal users of the various kmap interfaces, where the backing storage must be pinned to guarantee that the atomic kmap calls can succeed. Since there's no in-kernel users of the kmap interfaces yet this isn't a real problem.

Returns:

0 on success or a negative error code on failure. This can for example fail when the backing storage can't be allocated. Can also return -ERESTARTSYS or -EINTR when the call has been interrupted and needs to be restarted.

end_cpu_access This is called from *dma_buf_end_cpu_access()* when the importer is done accessing the CPU. The exporter can use this to flush caches and unpin any resources pinned in **begin_cpu_access**. The result of any dma_buf kmap calls after end_cpu_access is undefined.

This callback is optional.

Returns:

0 on success or a negative error code on failure. Can return -ERESTARTSYS or -EINTR when the call has been interrupted and needs to be restarted.

map_atomic maps a page from the buffer into kernel address space, users may not block until the subsequent unmap call. This callback must not sleep.

unmap_atomic [optional] unmaps a atomically mapped page from the buffer. This Callback must not sleep.

map maps a page from the buffer into kernel address space.

unmap [optional] unmaps a page from the buffer.

mmap This callback is used by the *dma_buf_mmap()* function

Note that the mapping needs to be incoherent, userspace is expected to braket CPU access using the DMA_BUF_IOCTL_SYNC interface.

Because dma-buf buffers have invariant size over their lifetime, the dma-buf core checks whether a vma is too large and rejects such mappings. The exporter hence does not need to duplicate this check. Drivers do not need to check this themselves.

If an exporter needs to manually flush caches and hence needs to fake coherency for mmap support, it needs to be able to zap all the ptes pointing at the backing storage. Now linux mm needs a struct address_space associated with the struct file stored in vma->vm_file to do that with the function unmap_mapping_range. But the dma_buf framework only backs every dma_buf fd with the anon_file struct file, i.e. all dma_bufs share the same file.

Hence exporters need to setup their own file (and address_space) association by setting vma->vm_file and adjusting vma->vm_pgoff in the dma_buf mmap callback. In the specific case of a gem driver the exporter could use the shmem file already provided by gem (and set vm_pgoff = 0). Exporters can then zap ptes by unmapping the corresponding range of the struct address_space associated with their own file.

This callback is optional.

Returns:

0 on success or a negative error code on failure.

vmap [optional] creates a virtual mapping for the buffer into kernel address space. Same restrictions as for vmap and friends apply.

vunmap [optional] unmaps a vmap from the buffer

struct **dma_buf**
shared buffer object

Definition

```
struct dma_buf {
  size_t size;
  struct file *file;
  struct list_head attachments;
  const struct dma_buf_ops *ops;
  struct mutex lock;
  unsigned vmapping_counter;
  void *vmap_ptr;
  const char *exp_name;
  struct module *owner;
  struct list_head list_node;
  void *priv;
  struct reservation_object *resv;
  wait_queue_head_t poll;
  struct dma_buf_poll_cb_t {
    struct dma_fence_cb cb;
    wait_queue_head_t *poll;
    __poll_t active;
  } cb_excl, cb_shared;
};
```

Members

size size of the buffer

file file pointer used for sharing buffers across, and for refcounting.

attachments list of dma_buf_attachment that denotes all devices attached.

ops dma_buf_ops associated with this buffer object.

lock used internally to serialize list manipulation, attach/detach and vmap/unmap

vmapping_counter used internally to refcnt the vmaps

vmap_ptr the current vmap ptr if vmapping_counter > 0

exp_name name of the exporter; useful for debugging.

owner pointer to exporter module; used for refcounting when exporter is a kernel module.

list_node node for dma_buf accounting and debugging.

priv exporter specific private data for this buffer object.

resv reservation object linked to this dma-buf

poll for userspace poll support

cb_excl for userspace poll support

cb_shared for userspace poll support

Description

This represents a shared buffer, created by calling *dma_buf_export()*. The userspace representation is a normal file descriptor, which can be created by calling *dma_buf_fd()*.

Shared dma buffers are reference counted using *dma_buf_put()* and *get_dma_buf()*.

Device DMA access is handled by the separate *struct dma_buf_attachment*.

struct **dma_buf_attachment**
holds device-buffer attachment data

Definition

```
struct dma_buf_attachment {
  struct dma_buf *dmabuf;
  struct device *dev;
  struct list_head node;
  void *priv;
};
```

Members

dmabuf buffer for this attachment.

dev device attached to the buffer.

node list of dma_buf_attachment.

priv exporter specific attachment data.

Description

This structure holds the attachment information between the dma_buf buffer and its user device(s). The list contains one attachment struct per device attached to the buffer.

An attachment is created by calling *dma_buf_attach()*, and released again by calling *dma_buf_detach()*. The DMA mapping itself needed to initiate a transfer is created by *dma_buf_map_attachment()* and freed again by calling *dma_buf_unmap_attachment()*.

struct **dma_buf_export_info**
holds information needed to export a dma_buf

Definition

```
struct dma_buf_export_info {
  const char *exp_name;
  struct module *owner;
  const struct dma_buf_ops *ops;
  size_t size;
  int flags;
  struct reservation_object *resv;
  void *priv;
};
```

Members

exp_name name of the exporter - useful for debugging.

owner pointer to exporter module - used for refcounting kernel module

ops Attach allocator-defined dma buf ops to the new buffer

size Size of the buffer

flags mode flags for the file

resv reservation-object, NULL to allocate default one

priv Attach private data of allocator to this buffer

Description

This structure holds the information required to export the buffer. Used with *dma_buf_export()* only.

DEFINE_DMA_BUF_EXPORT_INFO(*name*)
helper macro for exporters

Parameters

name export-info name

Description

DEFINE_DMA_BUF_EXPORT_INFO macro defines the *struct dma_buf_export_info*, zeroes it out and pre-populates exp_name in it.

void **get_dma_buf**(struct *dma_buf* * *dmabuf*)
convenience wrapper for get_file.

Parameters

struct dma_buf * dmabuf [in] pointer to dma_buf

Description

Increments the reference count on the dma-buf, needed in case of drivers that either need to create additional references to the dmabuf on the kernel side. For example, an exporter that needs to keep a dmabuf ptr so that subsequent exports don't create a new dmabuf.

Reservation Objects

The reservation object provides a mechanism to manage shared and exclusive fences associated with a buffer. A reservation object can have attached one exclusive fence (normally associated with write operations) or N shared fences (read operations). The RCU mechanism is used to protect read access to fences from locked write-side updates.

int **reservation_object_reserve_shared**(struct *reservation_object* * *obj*)
Reserve space to add a shared fence to a reservation_object.

Parameters

struct reservation_object * obj reservation object

Description

Should be called before `reservation_object_add_shared_fence()`. Must be called with obj->lock held.

RETURNS Zero for success, or -errno

void **reservation_object_add_shared_fence**(struct *reservation_object* * *obj*, struct *dma_fence* * *fence*)

Add a fence to a shared slot

Parameters

struct reservation_object * obj the reservation object

struct dma_fence * fence the shared fence to add

Description

Add a fence to a shared slot, obj->lock must be held, and `reservation_object_reserve_shared()` has been called.

void **reservation_object_add_excl_fence**(struct *reservation_object* * *obj*, struct *dma_fence* * *fence*)

Add an exclusive fence.

Parameters

struct reservation_object * obj the reservation object

struct dma_fence * fence the shared fence to add

Description

Add a fence to the exclusive slot. The obj->lock must be held.

int **reservation_object_copy_fences**(struct *reservation_object* * *dst*, struct *reservation_object* * *src*)

Copy all fences from src to dst.

Parameters

struct reservation_object * dst the destination reservation object

struct reservation_object * src the source reservation object

Description

Copy all fences from src to dst. dst-lock must be held.

int **reservation_object_get_fences_rcu**(struct *reservation_object* * *obj*, struct *dma_fence* ** *pfence_excl*, unsigned * *pshared_count*, struct *dma_fence* *** *pshared*)

Get an object's shared and exclusive fences without update side lock held

Parameters

struct reservation_object * obj the reservation object

struct dma_fence ** pfence_excl the returned exclusive fence (or NULL)

unsigned * pshared_count the number of shared fences returned

struct dma_fence * pshared** the array of shared fence ptrs returned (array is krealloc'd to the required size, and must be freed by caller)

Description

RETURNS Zero or -errno

long **reservation_object_wait_timeout_rcu**(struct *reservation_object* * *obj*, bool *wait_all*, bool *intr*, unsigned long *timeout*)

Wait on reservation's objects shared and/or exclusive fences.

Parameters

struct reservation_object * obj the reservation object

bool wait_all if true, wait on all fences, else wait on just exclusive fence

bool intr if true, do interruptible wait

unsigned long timeout timeout value in jiffies or zero to return immediately

Description

RETURNS Returns -ERESTARTSYS if interrupted, 0 if the wait timed out, or greater than zer on success.

bool **reservation_object_test_signaled_rcu**(struct *reservation_object* * *obj*, bool *test_all*)
Test if a reservation object's fences have been signaled.

Parameters

struct reservation_object * obj the reservation object

bool test_all if true, test all fences, otherwise only test the exclusive fence

Description

RETURNS true if all fences signaled, else false

struct **reservation_object_list**
a list of shared fences

Definition

```
struct reservation_object_list {
  struct rcu_head rcu;
  u32 shared_count, shared_max;
  struct dma_fence __rcu *shared[];
};
```

Members

rcu for internal use

shared_count table of shared fences

shared_max for growing shared fence table

shared shared fence table

struct **reservation_object**
a reservation object manages fences for a buffer

Definition

```
struct reservation_object {
  struct ww_mutex lock;
  seqcount_t seq;
  struct dma_fence __rcu *fence_excl;
  struct reservation_object_list __rcu *fence;
  struct reservation_object_list *staged;
};
```

Members

lock update side lock

seq sequence count for managing RCU read-side synchronization

fence_excl the exclusive fence, if there is one currently

fence list of current shared fences

staged staged copy of shared fences for RCU updates

void **reservation_object_init**(struct *reservation_object* * *obj*)
initialize a reservation object

Parameters

struct reservation_object * obj the reservation object

void **reservation_object_fini**(struct *reservation_object* * *obj*)
destroys a reservation object

Parameters

struct reservation_object * obj the reservation object

struct *reservation_object_list* * **reservation_object_get_list**(struct *reservation_object* * *obj*)
get the reservation object's shared fence list, with update-side lock held

Parameters

struct reservation_object * obj the reservation object

Description

Returns the shared fence list. Does NOT take references to the fence. The obj->lock must be held.

int **reservation_object_lock**(struct *reservation_object* * *obj*, struct ww_acquire_ctx * *ctx*)
lock the reservation object

Parameters

struct reservation_object * obj the reservation object

struct ww_acquire_ctx * ctx the locking context

Description

Locks the reservation object for exclusive access and modification. Note, that the lock is only against other writers, readers will run concurrently with a writer under RCU. The seqlock is used to notify readers if they overlap with a writer.

As the reservation object may be locked by multiple parties in an undefined order, a #ww_acquire_ctx is passed to unwind if a cycle is detected. See `ww_mutex_lock()` and `ww_acquire_init()`. A reservation object may be locked by itself by passing NULL as **ctx**.

int **reservation_object_lock_interruptible**(struct *reservation_object* * *obj*, struct ww_acquire_ctx * *ctx*)
lock the reservation object

Parameters

struct reservation_object * obj the reservation object

struct ww_acquire_ctx * ctx the locking context

Description

Locks the reservation object interruptible for exclusive access and modification. Note, that the lock is only against other writers, readers will run concurrently with a writer under RCU. The seqlock is used to notify readers if they overlap with a writer.

As the reservation object may be locked by multiple parties in an undefined order, a #ww_acquire_ctx is passed to unwind if a cycle is detected. See `ww_mutex_lock()` and `ww_acquire_init()`. A reservation object may be locked by itself by passing NULL as **ctx**.

bool **reservation_object_trylock**(struct *reservation_object* * *obj*)
trylock the reservation object

Parameters

struct reservation_object * obj the reservation object

Description

Tries to lock the reservation object for exclusive access and modification. Note, that the lock is only against other writers, readers will run concurrently with a writer under RCU. The seqlock is used to notify readers if they overlap with a writer.

Also note that since no context is provided, no deadlock protection is possible.

Returns true if the lock was acquired, false otherwise.

void **reservation_object_unlock**(struct *reservation_object* * *obj*)
unlock the reservation object

Parameters

struct reservation_object * obj the reservation object

Description

Unlocks the reservation object following exclusive access.

struct *dma_fence* * **reservation_object_get_excl**(struct *reservation_object* * *obj*)
get the reservation object's exclusive fence, with update-side lock held

Parameters

struct reservation_object * obj the reservation object

Description

Returns the exclusive fence (if any). Does NOT take a reference. The obj->lock must be held.

RETURNS The exclusive fence or NULL

struct *dma_fence* * **reservation_object_get_excl_rcu**(struct *reservation_object* * *obj*)
get the reservation object's exclusive fence, without lock held.

Parameters

struct reservation_object * obj the reservation object

Description

If there is an exclusive fence, this atomically increments it's reference count and returns it.

RETURNS The exclusive fence or NULL if none

DMA Fences

u64 **dma_fence_context_alloc**(unsigned *num*)
allocate an array of fence contexts

Parameters

unsigned 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.

int **dma_fence_signal_locked**(struct *dma_fence* * *fence*)
signal completion of a fence

Parameters

struct dma_fence * fence the fence to signal

Description

Signal completion for software callbacks on a fence, this will unblock *dma_fence_wait()* calls and run all the callbacks added with *dma_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 dma_fence_signal, this function must be called with fence->lock held.

int **dma_fence_signal**(struct *dma_fence* * *fence*)
signal completion of a fence

Parameters

struct dma_fence * fence the fence to signal

Description

Signal completion for software callbacks on a fence, this will unblock *dma_fence_wait()* calls and run all the callbacks added with *dma_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.

signed long **dma_fence_wait_timeout**(struct *dma_fence* * *fence*, bool *intr*, signed long *timeout*)
sleep until the fence gets signaled or until timeout elapses

Parameters

struct dma_fence * fence [in] the fence to wait on

bool intr [in] if true, do an interruptible wait

signed long 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.

void **dma_fence_enable_sw_signaling**(struct *dma_fence* * *fence*)
enable signaling on fence

Parameters

struct dma_fence * 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

int **dma_fence_add_callback**(struct *dma_fence* * *fence*, struct *dma_fence_cb* * *cb*, dma_fence_func_t *func*)
add a callback to be called when the fence is signaled

Parameters

struct dma_fence * fence [in] the fence to wait on

struct dma_fence_cb * cb [in] the callback to register

dma_fence_func_t func [in] the function to call

Description

cb will be initialized by dma_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 dma_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 dma_fence_signal. The callback itself can be called from irq context.

Returns 0 in case of success, -ENOENT if the fence is already signaled and -EINVAL in case of error.

int **dma_fence_get_status**(struct *dma_fence* * *fence*)
returns the status upon completion

Parameters

struct dma_fence * fence [in] the dma_fence to query

Description

This wraps *dma_fence_get_status_locked()* to return the error status condition on a signaled fence. See *dma_fence_get_status_locked()* for more details.

Returns 0 if the fence has not yet been signaled, 1 if the fence has been signaled without an error condition, or a negative error code if the fence has been completed in err.

bool **dma_fence_remove_callback**(struct *dma_fence* * *fence*, struct *dma_fence_cb* * *cb*)
remove a callback from the signaling list

Parameters

struct dma_fence * fence [in] the fence to wait on

struct dma_fence_cb * 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.

signed long **dma_fence_default_wait**(struct *dma_fence* * *fence*, bool *intr*, signed long *timeout*)
default sleep until the fence gets signaled or until timeout elapses

Parameters

struct dma_fence * fence [in] the fence to wait on

bool intr [in] if true, do an interruptible wait

signed long 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. If timeout is zero the value one is returned if the fence is already signaled for consistency with other functions taking a jiffies timeout.

signed long **dma_fence_wait_any_timeout**(struct *dma_fence* ** *fences*, uint32_t *count*, bool *intr*, signed long *timeout*, uint32_t * *idx*)
sleep until any fence gets signaled or until timeout elapses

Parameters

struct dma_fence ** fences [in] array of fences to wait on

uint32_t count [in] number of fences to wait on

bool intr [in] if true, do an interruptible wait

signed long timeout [in] timeout value in jiffies, or MAX_SCHEDULE_TIMEOUT

uint32_t * idx [out] the first signaled fence index, meaningful only on positive return

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.

void **dma_fence_init**(struct *dma_fence* * *fence*, const struct *dma_fence_ops* * *ops*, spinlock_t * *lock*, u64 *context*, unsigned *seqno*)

Initialize a custom fence.

Parameters

struct dma_fence * fence [in] the fence to initialize

const struct dma_fence_ops * ops [in] the dma_fence_ops for operations on this fence

spinlock_t * lock [in] the irqsafe spinlock to use for locking this fence

u64 context [in] the execution context this fence is run on

unsigned seqno [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 dma_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 dma_fence_later.

struct **dma_fence**

software synchronization primitive

Definition

```
struct dma_fence {
  struct kref refcount;
  const struct dma_fence_ops *ops;
  struct rcu_head rcu;
  struct list_head cb_list;
  spinlock_t *lock;
  u64 context;
  unsigned seqno;
  unsigned long flags;
  ktime_t timestamp;
  int error;
};
```

Members

refcount refcount for this fence

ops dma_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 *dma_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 DMA_FENCE_FLAG_* defined below

timestamp Timestamp when the fence was signaled.

error Optional, only valid if < 0, must be set before calling dma_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.

DMA_FENCE_FLAG_SIGNALED_BIT - fence is already signaled DMA_FENCE_FLAG_TIMESTAMP_BIT - timestamp recorded for fence signaling DMA_FENCE_FLAG_ENABLE_SIGNAL_BIT - enable_signaling might have been called DMA_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 dma_fence_signal was called right before this bit was set, it would have been able to set the DMA_FENCE_FLAG_SIGNALED_BIT, before enable_signaling was called. Adding a check for DMA_FENCE_FLAG_SIGNALED_BIT after setting DMA_FENCE_FLAG_ENABLE_SIGNAL_BIT closes this race, and makes sure that after dma_fence_signal was called, any enable_signaling call will have either been completed, or never called at all.

struct **dma_fence_cb**
callback for dma_fence_add_callback

Definition

```
struct dma_fence_cb {
  struct list_head node;
  dma_fence_func_t func;
};
```

Members

node used by dma_fence_add_callback to append this struct to fence::cb_list

func dma_fence_func_t to call

Description

This struct will be initialized by dma_fence_add_callback, additional data can be passed along by embedding dma_fence_cb in another struct.

struct **dma_fence_ops**
operations implemented for fence

Definition

```
struct dma_fence_ops {
  const char * (*get_driver_name)(struct dma_fence *fence);
  const char * (*get_timeline_name)(struct dma_fence *fence);
  bool (*enable_signaling)(struct dma_fence *fence);
  bool (*signaled)(struct dma_fence *fence);
  signed long (*wait)(struct dma_fence *fence, bool intr, signed long timeout);
  void (*release)(struct dma_fence *fence);
  int (*fill_driver_data)(struct dma_fence *fence, void *data, int size);
  void (*fence_value_str)(struct dma_fence *fence, char *str, int size);
  void (*timeline_value_str)(struct dma_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 dma_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

Description

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 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->error may be set in enable_signaling, but only when false is returned.

Calling dma_fence_signal before enable_signaling is called allows for a tiny race window in which enable_signaling is called during, before, or after dma_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 dma_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->error if returning true.

Notes on wait: Must not be NULL, set to dma_fence_default_wait for default implementation. the dma_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.

void **dma_fence_put**(struct *dma_fence* * *fence*)
    decreases refcount of the fence

Parameters

struct dma_fence * fence [in] fence to reduce refcount of

struct *dma_fence* * **dma_fence_get**(struct *dma_fence* * *fence*)
    increases refcount of the fence

Parameters

struct dma_fence * fence [in] fence to increase refcount of

Description

Returns the same fence, with refcount increased by 1.

struct *dma_fence* * **dma_fence_get_rcu**(struct *dma_fence* * *fence*)
    get a fence from a reservation_object_list with rcu read lock

Parameters

struct dma_fence * fence [in] fence to increase refcount of

Description

Function returns NULL if no refcount could be obtained, or the fence.

struct *dma_fence* * **dma_fence_get_rcu_safe**(struct *dma_fence* __rcu ** *fencep*)
acquire a reference to an RCU tracked fence

Parameters

struct dma_fence __rcu ** fencep [in] pointer to fence to increase refcount of

Description

Function returns NULL if no refcount could be obtained, or the fence. This function handles acquiring a reference to a fence that may be reallocated within the RCU grace period (such as with SLAB_TYPESAFE_BY_RCU), so long as the caller is using RCU on the pointer to the fence.

An alternative mechanism is to employ a seqlock to protect a bunch of fences, such as used by struct reservation_object. When using a seqlock, the seqlock must be taken before and checked after a reference to the fence is acquired (as shown here).

The caller is required to hold the RCU read lock.

bool **dma_fence_is_signaled_locked**(struct *dma_fence* * *fence*)
Return an indication if the fence is signaled yet.

Parameters

struct dma_fence * 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 dma_fence_add_callback, dma_fence_wait or dma_fence_enable_sw_signaling haven't been called before.

This function requires fence->lock to be held.

bool **dma_fence_is_signaled**(struct *dma_fence* * *fence*)
Return an indication if the fence is signaled yet.

Parameters

struct dma_fence * 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 dma_fence_add_callback, dma_fence_wait or dma_fence_enable_sw_signaling haven't been called before.

It's recommended for seqno fences to call dma_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.

bool **__dma_fence_is_later**(u32 *f1*, u32 *f2*)
return if f1 is chronologically later than f2

Parameters

u32 f1 [in] the first fence's seqno

u32 f2 [in] the second fence's seqno 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 common across contexts.

bool **dma_fence_is_later**(struct *dma_fence* * *f1*, struct *dma_fence* * *f2*)
return if f1 is chronologically later than f2

Parameters

struct dma_fence * f1 [in] the first fence from the same context

struct dma_fence * 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.

struct *dma_fence* * **dma_fence_later**(struct *dma_fence* * *f1*, struct *dma_fence* * *f2*)
 return the chronologically later fence

Parameters

struct dma_fence * f1 [in] the first fence from the same context

struct dma_fence * 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.

int **dma_fence_get_status_locked**(struct *dma_fence* * *fence*)
 returns the status upon completion

Parameters

struct dma_fence * fence [in] the dma_fence to query

Description

Drivers can supply an optional error status condition before they signal the fence (to indicate whether the fence was completed due to an error rather than success). The value of the status condition is only valid if the fence has been signaled, `dma_fence_get_status_locked()` first checks the signal state before reporting the error status.

Returns 0 if the fence has not yet been signaled, 1 if the fence has been signaled without an error condition, or a negative error code if the fence has been completed in err.

void **dma_fence_set_error**(struct *dma_fence* * *fence*, int *error*)
 flag an error condition on the fence

Parameters

struct dma_fence * fence [in] the dma_fence

int error [in] the error to store

Description

Drivers can supply an optional error status condition before they signal the fence, to indicate that the fence was completed due to an error rather than success. This must be set before signaling (so that the value is visible before any waiters on the signal callback are woken). This helper exists to help catching erroneous setting of #dma_fence.error.

signed long **dma_fence_wait**(struct *dma_fence* * *fence*, bool *intr*)
 sleep until the fence gets signaled

Parameters

struct dma_fence * fence [in] the fence to wait on

bool 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.

Seqno Hardware Fences

struct seqno_fence * **to_seqno_fence**(struct *dma_fence* * *fence*)
cast a fence to a seqno_fence

Parameters

struct dma_fence * fence fence to cast to a seqno_fence

Description

Returns NULL if the fence is not a seqno_fence, or the seqno_fence otherwise.

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 *dma_fence_ops* * *ops*)
initialize a seqno fence

Parameters

struct seqno_fence * fence seqno_fence to initialize

spinlock_t * lock pointer to spinlock to use for fence

struct dma_buf * sync_buf buffer containing the memory location to signal on

uint32_t context the execution context this fence is a part of

uint32_t seqno_ofs the offset within **sync_buf**

uint32_t seqno the sequence # to signal on

enum seqno_fence_condition cond fence wait condition

const struct dma_fence_ops * ops 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 *dma_fence_signal()* before destruction. This will prevent possible issues from wraparound at time of issue vs time of check, since users can check *dma_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.

DMA Fence Array

struct *dma_fence_array* * **dma_fence_array_create**(int *num_fences*, struct *dma_fence* ** *fences*, u64 *context*, unsigned *seqno*, bool *signal_on_any*)
Create a custom fence array

Parameters

int num_fences [in] number of fences to add in the array

struct dma_fence ** fences [in] array containing the fences

u64 context [in] fence context to use

unsigned seqno [in] sequence number to use

bool signal_on_any [in] signal on any fence in the array

Description

Allocate a dma_fence_array object and initialize the base fence with *dma_fence_init()*. In case of error it returns NULL.

The caller should allocate the fences array with num_fences size and fill it with the fences it wants to add to the object. Ownership of this array is taken and *dma_fence_put()* is used on each fence on release.

If **signal_on_any** is true the fence array signals if any fence in the array signals, otherwise it signals when all fences in the array signal.

bool **dma_fence_match_context**(struct *dma_fence* * *fence*, u64 *context*)
Check if all fences are from the given context

Parameters

struct dma_fence * fence [in] fence or fence array

u64 context [in] fence context to check all fences against

Description

Checks the provided fence or, for a fence array, all fences in the array against the given context. Returns false if any fence is from a different context.

struct **dma_fence_array_cb**
callback helper for fence array

Definition

```
struct dma_fence_array_cb {
  struct dma_fence_cb cb;
  struct dma_fence_array *array;
};
```

Members

cb fence callback structure for signaling

array reference to the parent fence array object

struct **dma_fence_array**
fence to represent an array of fences

Definition

```
struct dma_fence_array {
  struct dma_fence base;
  spinlock_t lock;
  unsigned num_fences;
  atomic_t num_pending;
  struct dma_fence **fences;
  struct irq_work work;
};
```

Members

base fence base class

lock spinlock for fence handling

num_fences number of fences in the array

num_pending fences in the array still pending

fences array of the fences

bool **dma_fence_is_array**(struct *dma_fence* * *fence*)
check if a fence is from the array subsclass

Parameters

struct dma_fence * fence fence to test

Description

Return true if it is a dma_fence_array and false otherwise.

struct *dma_fence_array* * **to_dma_fence_array**(struct *dma_fence* * *fence*)
cast a fence to a dma_fence_array

Parameters

struct dma_fence * fence fence to cast to a dma_fence_array

Description

Returns NULL if the fence is not a dma_fence_array, or the dma_fence_array otherwise.

DMA Fence uABI/Sync File

struct *sync_file* * **sync_file_create**(struct *dma_fence* * *fence*)
creates a sync file

Parameters

struct dma_fence * fence fence to add to the sync_fence

Description

Creates a sync_file containg **fence**. This function acquires and additional reference of **fence** for the newly-created *sync_file*, if it succeeds. The sync_file can be released with fput(sync_file->file). Returns the sync_file or NULL in case of error.

struct *dma_fence* * **sync_file_get_fence**(int *fd*)
get the fence related to the sync_file fd

Parameters

int fd sync_file fd to get the fence from

Description

Ensures **fd** references a valid sync_file and returns a fence that represents all fence in the sync_file. On error NULL is returned.

struct **sync_file**
sync file to export to the userspace

Definition

```
struct sync_file {
  struct file             *file;
  char user_name[32];
#ifdef CONFIG_DEBUG_FS;
  struct list_head        sync_file_list;
#endif;
  wait_queue_head_t wq;
  unsigned long           flags;
  struct dma_fence        *fence;
  struct dma_fence_cb cb;
};
```

Members

`file` file representing this fence

`user_name` Name of the sync file provided by userspace, for merged fences. Otherwise generated through driver callbacks (in which case the entire array is 0).

`sync_file_list` membership in global file list

`wq` wait queue for fence signaling

`flags` flags for the sync_file

`fence` fence with the fences in the sync_file

`cb` fence callback information

Description

flags: POLL_ENABLED: whether userspace is currently `poll()`‘ing or not

CHAPTER

SIX

DEVICE LINKS

By default, the driver core only enforces dependencies between devices that are borne out of a parent/child relationship within the device hierarchy: When suspending, resuming or shutting down the system, devices are ordered based on this relationship, i.e. children are always suspended before their parent, and the parent is always resumed before its children.

Sometimes there is a need to represent device dependencies beyond the mere parent/child relationship, e.g. between siblings, and have the driver core automatically take care of them.

Secondly, the driver core by default does not enforce any driver presence dependencies, i.e. that one device must be bound to a driver before another one can probe or function correctly.

Often these two dependency types come together, so a device depends on another one both with regards to driver presence *and* with regards to suspend/resume and shutdown ordering.

Device links allow representation of such dependencies in the driver core.

In its standard form, a device link combines *both* dependency types: It guarantees correct suspend/resume and shutdown ordering between a "supplier" device and its "consumer" devices, and it guarantees driver presence on the supplier. The consumer devices are not probed before the supplier is bound to a driver, and they're unbound before the supplier is unbound.

When driver presence on the supplier is irrelevant and only correct suspend/resume and shutdown ordering is needed, the device link may simply be set up with the `DL_FLAG_STATELESS` flag. In other words, enforcing driver presence on the supplier is optional.

Another optional feature is runtime PM integration: By setting the `DL_FLAG_PM_RUNTIME` flag on addition of the device link, the PM core is instructed to runtime resume the supplier and keep it active whenever and for as long as the consumer is runtime resumed.

Usage

The earliest point in time when device links can be added is after *`device_add()`* has been called for the supplier and *`device_initialize()`* has been called for the consumer.

It is legal to add them later, but care must be taken that the system remains in a consistent state: E.g. a device link cannot be added in the midst of a suspend/resume transition, so either commencement of such a transition needs to be prevented with `lock_system_sleep()`, or the device link needs to be added from a function which is guaranteed not to run in parallel to a suspend/resume transition, such as from a device `->probe` callback or a boot-time PCI quirk.

Another example for an inconsistent state would be a device link that represents a driver presence dependency, yet is added from the consumer's `->probe` callback while the supplier hasn't probed yet: Had the driver core known about the device link earlier, it wouldn't have probed the consumer in the first place. The onus is thus on the consumer to check presence of the supplier after adding the link, and defer probing on non-presence.

If a device link is added in the `->probe` callback of the supplier or consumer driver, it is typically deleted in its `->remove` callback for symmetry. That way, if the driver is compiled as a module, the device link is added on module load and orderly deleted on unload. The same restrictions that apply to device link addition (e.g. exclusion of a parallel suspend/resume transition) apply equally to deletion.

Several flags may be specified on device link addition, two of which have already been mentioned above: `DL_FLAG_STATELESS` to express that no driver presence dependency is needed (but only correct suspend/resume and shutdown ordering) and `DL_FLAG_PM_RUNTIME` to express that runtime PM integration is desired.

Two other flags are specifically targeted at use cases where the device link is added from the consumer's `->probe` callback: `DL_FLAG_RPM_ACTIVE` can be specified to runtime resume the supplier upon addition of the device link. `DL_FLAG_AUTOREMOVE` causes the device link to be automatically purged when the consumer fails to probe or later unbinds. This obviates the need to explicitly delete the link in the `->remove` callback or in the error path of the `->probe` callback.

Limitations

Driver authors should be aware that a driver presence dependency (i.e. when `DL_FLAG_STATELESS` is not specified on link addition) may cause probing of the consumer to be deferred indefinitely. This can become a problem if the consumer is required to probe before a certain initcall level is reached. Worse, if the supplier driver is blacklisted or missing, the consumer will never be probed.

Sometimes drivers depend on optional resources. They are able to operate in a degraded mode (reduced feature set or performance) when those resources are not present. An example is an SPI controller that can use a DMA engine or work in PIO mode. The controller can determine presence of the optional resources at probe time but on non-presence there is no way to know whether they will become available in the near future (due to a supplier driver probing) or never. Consequently it cannot be determined whether to defer probing or not. It would be possible to notify drivers when optional resources become available after probing, but it would come at a high cost for drivers as switching between modes of operation at runtime based on the availability of such resources would be much more complex than a mechanism based on probe deferral. In any case optional resources are beyond the scope of device links.

Examples

- An MMU device exists alongside a busmaster device, both are in the same power domain. The MMU implements DMA address translation for the busmaster device and shall be runtime resumed and kept active whenever and as long as the busmaster device is active. The busmaster device's driver shall not bind before the MMU is bound. To achieve this, a device link with runtime PM integration is added from the busmaster device (consumer) to the MMU device (supplier). The effect with regards to runtime PM is the same as if the MMU was the parent of the master device.

  The fact that both devices share the same power domain would normally suggest usage of a *`struct dev_pm_domain`* or `struct generic_pm_domain`, however these are not independent devices that happen to share a power switch, but rather the MMU device serves the busmaster device and is useless without it. A device link creates a synthetic hierarchical relationship between the devices and is thus more apt.

- A Thunderbolt host controller comprises a number of PCIe hotplug ports and an NHI device to manage the PCIe switch. On resume from system sleep, the NHI device needs to re-establish PCI tunnels to attached devices before the hotplug ports can resume. If the hotplug ports were children of the NHI, this resume order would automatically be enforced by the PM core, but unfortunately they're aunts. The solution is to add device links from the hotplug ports (consumers) to the NHI device (supplier). A driver presence dependency is not necessary for this use case.

- Discrete GPUs in hybrid graphics laptops often feature an HDA controller for HDMI/DP audio. In the device hierarchy the HDA controller is a sibling of the VGA device, yet both share the same power domain and the HDA controller is only ever needed when an HDMI/DP display is attached to the VGA device. A device link from the HDA controller (consumer) to the VGA device (supplier) aptly represents this relationship.

- ACPI allows definition of a device start order by way of _DEP objects. A classical example is when ACPI power management methods on one device are implemented in terms of I^{2}C accesses and require a specific I^{2}C controller to be present and functional for the power management of the device in question to work.
- In some SoCs a functional dependency exists from display, video codec and video processing IP cores on transparent memory access IP cores that handle burst access and compression/decompression.

Alternatives

- A *struct dev_pm_domain* can be used to override the bus, class or device type callbacks. It is intended for devices sharing a single on/off switch, however it does not guarantee a specific suspend/resume ordering, this needs to be implemented separately. It also does not by itself track the runtime PM status of the involved devices and turn off the power switch only when all of them are runtime suspended. Furthermore it cannot be used to enforce a specific shutdown ordering or a driver presence dependency.
- A `struct generic_pm_domain` is a lot more heavyweight than a device link and does not allow for shutdown ordering or driver presence dependencies. It also cannot be used on ACPI systems.

Implementation

The device hierarchy, which – as the name implies – is a tree, becomes a directed acyclic graph once device links are added.

Ordering of these devices during suspend/resume is determined by the dpm_list. During shutdown it is determined by the devices_kset. With no device links present, the two lists are a flattened, one-dimensional representations of the device tree such that a device is placed behind all its ancestors. That is achieved by traversing the ACPI namespace or OpenFirmware device tree top-down and appending devices to the lists as they are discovered.

Once device links are added, the lists need to satisfy the additional constraint that a device is placed behind all its suppliers, recursively. To ensure this, upon addition of the device link the consumer and the entire sub-graph below it (all children and consumers of the consumer) are moved to the end of the list. (Call to `device_reorder_to_tail()` from *device_link_add()*.)

To prevent introduction of dependency loops into the graph, it is verified upon device link addition that the supplier is not dependent on the consumer or any children or consumers of the consumer. (Call to `device_is_dependent()` from *device_link_add()*.) If that constraint is violated, *device_link_add()* will return `NULL` and a `WARNING` will be logged.

Notably this also prevents the addition of a device link from a parent device to a child. However the converse is allowed, i.e. a device link from a child to a parent. Since the driver core already guarantees correct suspend/resume and shutdown ordering between parent and child, such a device link only makes sense if a driver presence dependency is needed on top of that. In this case driver authors should weigh carefully if a device link is at all the right tool for the purpose. A more suitable approach might be to simply use deferred probing or add a device flag causing the parent driver to be probed before the child one.

State machine

enum **device_link_state**
Device link states.

Constants

DL_STATE_NONE The presence of the drivers is not being tracked.

DL_STATE_DORMANT None of the supplier/consumer drivers is present.

DL_STATE_AVAILABLE The supplier driver is present, but the consumer is not.

DL_STATE_CONSUMER_PROBE The consumer is probing (supplier driver present).

DL_STATE_ACTIVE Both the supplier and consumer drivers are present.

DL_STATE_SUPPLIER_UNBIND The supplier driver is unbinding.

```
                 .=============================.
                 |                             |
                 v                             |
DORMANT <=> AVAILABLE <=> CONSUMER_PROBE => ACTIVE
   ^                                           |
   |                                           |
   '============ SUPPLIER_UNBIND <============'
```

- The initial state of a device link is automatically determined by *device_link_add()* based on the driver presence on the supplier and consumer. If the link is created before any devices are probed, it is set to DL_STATE_DORMANT.
- When a supplier device is bound to a driver, links to its consumers progress to DL_STATE_AVAILABLE. (Call to device_links_driver_bound() from driver_bound().)
- Before a consumer device is probed, presence of supplier drivers is verified by checking that links to suppliers are in DL_STATE_AVAILABLE state. The state of the links is updated to DL_STATE_CONSUMER_PROBE. (Call to device_links_check_suppliers() from really_probe().) This prevents the supplier from unbinding. (Call to *wait_for_device_probe()* from device_links_unbind_consumers().)
- If the probe fails, links to suppliers revert back to DL_STATE_AVAILABLE. (Call to device_links_no_driver() from really_probe().)
- If the probe succeeds, links to suppliers progress to DL_STATE_ACTIVE. (Call to device_links_driver_bound() from driver_bound().)
- When the consumer's driver is later on removed, links to suppliers revert back to DL_STATE_AVAILABLE. (Call to __device_links_no_driver() from device_links_driver_cleanup(), which in turn is called from __device_release_driver().)
- Before a supplier's driver is removed, links to consumers that are not bound to a driver are updated to DL_STATE_SUPPLIER_UNBIND. (Call to device_links_busy() from __device_release_driver().) This prevents the consumers from binding. (Call to device_links_check_suppliers() from really_probe().) Consumers that are bound are freed from their driver; consumers that are probing are waited for until they are done. (Call to device_links_unbind_consumers() from __device_release_driver().) Once all links to consumers are in DL_STATE_SUPPLIER_UNBIND state, the supplier driver is released and the links revert to DL_STATE_DORMANT. (Call to device_links_driver_cleanup() from __device_release_driver().)

API

struct *device_link* * **device_link_add**(struct *device* * *consumer*, struct *device* * *supplier*, u32 *flags*)
Create a link between two devices.

Parameters

struct device * consumer Consumer end of the link.

struct device * supplier Supplier end of the link.

u32 flags Link flags.

Description

The caller is responsible for the proper synchronization of the link creation with runtime PM. First, setting the DL_FLAG_PM_RUNTIME flag will cause the runtime PM framework to take the link into account. Second, if the DL_FLAG_RPM_ACTIVE flag is set in addition to it, the supplier devices will be forced into the active metastate and reference-counted upon the creation of the link. If DL_FLAG_PM_RUNTIME is not set, DL_FLAG_RPM_ACTIVE will be ignored.

If the DL_FLAG_AUTOREMOVE is set, the link will be removed automatically when the consumer device driver unbinds from it. The combination of both DL_FLAG_AUTOREMOVE and DL_FLAG_STATELESS set is invalid and will cause NULL to be returned.

A side effect of the link creation is re-ordering of dpm_list and the devices_kset list by moving the consumer device and all devices depending on it to the ends of these lists (that does not happen to devices that have not been registered when this function is called).

The supplier device is required to be registered when this function is called and NULL will be returned if that is not the case. The consumer device need not be registered, however.

void **device_link_del**(struct *device_link* * *link*)
Delete a link between two devices.

Parameters

struct device_link * link Device link to delete.

Description

The caller must ensure proper synchronization of this function with runtime PM.

CHAPTER

SEVEN

MESSAGE-BASED DEVICES

Fusion message devices

u8 **mpt_register**(MPT_CALLBACK *cbfunc*, MPT_DRIVER_CLASS *dclass*, char * *func_name*)
Register protocol-specific main callback handler.

Parameters

MPT_CALLBACK cbfunc callback function pointer

MPT_DRIVER_CLASS dclass Protocol driver's class (MPT_DRIVER_CLASS enum value)

char * func_name 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.

void **mpt_deregister**(u8 *cb_idx*)
Deregister a protocol drivers resources.

Parameters

u8 cb_idx previously registered callback handle

Description

Each protocol-specific driver should call this routine when its module is unloaded.

int **mpt_event_register**(u8 *cb_idx*, MPT_EVHANDLER *ev_cbfunc*)
Register protocol-specific event callback handler.

Parameters

u8 cb_idx previously registered (via mpt_register) callback handle

MPT_EVHANDLER 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.

void **mpt_event_deregister**(u8 *cb_idx*)
Deregister protocol-specific event callback handler

Parameters

u8 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.

int **mpt_reset_register**(u8 *cb_idx*, MPT_RESETHANDLER *reset_func*)
Register protocol-specific IOC reset handler.

Parameters

u8 cb_idx previously registered (via mpt_register) callback handle

MPT_RESETHANDLER 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.

void **mpt_reset_deregister**(u8 *cb_idx*)
Deregister protocol-specific IOC reset handler.

Parameters

u8 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.

int **mpt_device_driver_register**(struct mpt_pci_driver * *dd_cbfunc*, u8 *cb_idx*)
Register device driver hooks

Parameters

struct mpt_pci_driver * dd_cbfunc driver callbacks struct

u8 cb_idx MPT protocol driver index

void **mpt_device_driver_deregister**(u8 *cb_idx*)
DeRegister device driver hooks

Parameters

u8 cb_idx MPT protocol driver index

MPT_FRAME_HDR* **mpt_get_msg_frame**(u8 *cb_idx*, MPT_ADAPTER * *ioc*)
Obtain an MPT request frame from the pool

Parameters

u8 cb_idx Handle of registered MPT protocol driver

MPT_ADAPTER * 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.

void **mpt_put_msg_frame**(u8 *cb_idx*, MPT_ADAPTER * *ioc*, MPT_FRAME_HDR * *mf*)
Send a protocol-specific MPT request frame to an IOC

Parameters

`u8 cb_idx` Handle of registered MPT protocol driver

`MPT_ADAPTER * ioc` Pointer to MPT adapter structure

`MPT_FRAME_HDR * mf` Pointer to MPT request frame

Description

This routine posts an MPT request frame to the request post FIFO of a specific MPT adapter.

void **`mpt_put_msg_frame_hi_pri`**(u8 *cb_idx*, MPT_ADAPTER * *ioc*, MPT_FRAME_HDR * *mf*)
Send a hi-pri protocol-specific MPT request frame

Parameters

`u8 cb_idx` Handle of registered MPT protocol driver

`MPT_ADAPTER * ioc` Pointer to MPT adapter structure

`MPT_FRAME_HDR * 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.

void **`mpt_free_msg_frame`**(MPT_ADAPTER * *ioc*, MPT_FRAME_HDR * *mf*)
Place MPT request frame back on FreeQ.

Parameters

`MPT_ADAPTER * ioc` Pointer to MPT adapter structure

`MPT_FRAME_HDR * mf` Pointer to MPT request frame

Description

This routine places a MPT request frame back on the MPT adapter's FreeQ.

int **`mpt_send_handshake_request`**(u8 *cb_idx*, MPT_ADAPTER * *ioc*, int *reqBytes*, u32 * *req*, int *sleepFlag*)
Send MPT request via doorbell handshake method.

Parameters

`u8 cb_idx` Handle of registered MPT protocol driver

`MPT_ADAPTER * ioc` Pointer to MPT adapter structure

`int reqBytes` Size of the request in bytes

`u32 * req` Pointer to MPT request frame

`int sleepFlag` 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.

int **`mpt_verify_adapter`**(int *iocid*, MPT_ADAPTER ** *iocpp*)
Given IOC identifier, set pointer to its adapter structure.

Parameters

`int iocid` IOC unique identifier (integer)

`MPT_ADAPTER ** 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.

int **`mpt_attach`**(struct pci_dev * *pdev*, const struct pci_device_id * *id*)
Install a PCI intelligent MPT adapter.

Parameters

`struct pci_dev * pdev` Pointer to pci_dev structure

`const struct pci_device_id * 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

void **`mpt_detach`**(struct pci_dev * *pdev*)
Remove a PCI intelligent MPT adapter.

Parameters

`struct pci_dev * pdev` Pointer to pci_dev structure

int **`mpt_suspend`**(struct pci_dev * *pdev*, pm_message_t *state*)
Fusion MPT base driver suspend routine.

Parameters

`struct pci_dev * pdev` Pointer to pci_dev structure

`pm_message_t state` new state to enter

int **`mpt_resume`**(struct pci_dev * *pdev*)
Fusion MPT base driver resume routine.

Parameters

`struct pci_dev * pdev` Pointer to pci_dev structure

u32 **`mpt_GetIocState`**(MPT_ADAPTER * *ioc*, int *cooked*)
Get the current state of a MPT adapter.

Parameters

`MPT_ADAPTER * ioc` Pointer to MPT_ADAPTER structure

`int 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.

int **`mpt_alloc_fw_memory`**(MPT_ADAPTER * *ioc*, int *size*)
allocate firmware memory

Parameters

`MPT_ADAPTER * ioc` Pointer to MPT_ADAPTER structure

int 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

void **mpt_free_fw_memory**(MPT_ADAPTER * *ioc*)
free firmware memory

Parameters

MPT_ADAPTER * ioc Pointer to MPT_ADAPTER structure

Description

If alt_img is NULL, delete from ioc structure. Else, delete a secondary image in same format.

int **mptbase_sas_persist_operation**(MPT_ADAPTER * *ioc*, u8 *persist_opcode*)
Perform operation on SAS Persistent Table

Parameters

MPT_ADAPTER * ioc Pointer to MPT_ADAPTER structure

u8 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

int **mpt_raid_phys_disk_pg0**(MPT_ADAPTER * *ioc*, u8 *phys_disk_num*, RaidPhysDiskPage0_t * *phys_disk*)
returns phys disk page zero

Parameters

MPT_ADAPTER * ioc Pointer to a Adapter Structure

u8 phys_disk_num io unit unique phys disk num generated by the ioc

RaidPhysDiskPage0_t * phys_disk 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

int **mpt_raid_phys_disk_get_num_paths**(MPT_ADAPTER * *ioc*, u8 *phys_disk_num*)
returns number paths associated to this phys_num

Parameters

MPT_ADAPTER * ioc Pointer to a Adapter Structure

u8 phys_disk_num io unit unique phys disk num generated by the ioc

Return

returns number paths

int **mpt_raid_phys_disk_pg1**(MPT_ADAPTER * *ioc*, u8 *phys_disk_num*, RaidPhysDiskPage1_t * *phys_disk*)
returns phys disk page 1

Parameters

`MPT_ADAPTER * ioc` Pointer to a Adapter Structure

`u8 phys_disk_num` io unit unique phys disk num generated by the ioc

`RaidPhysDiskPage1_t * phys_disk` 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

int **`mpt_findImVolumes`**(MPT_ADAPTER * *ioc*)
Identify IDs of hidden disks and RAID Volumes

Parameters

`MPT_ADAPTER * 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

int **`mpt_config`**(MPT_ADAPTER * *ioc*, CONFIGPARMS * *pCfg*)
Generic function to issue config message

Parameters

`MPT_ADAPTER * ioc` Pointer to an adapter structure

`CONFIGPARMS * 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)

void **`mpt_print_ioc_summary`**(MPT_ADAPTER * *ioc*, char * *buffer*, int * *size*, int *len*, int *showlan*)
Write ASCII summary of IOC to a buffer.

Parameters

`MPT_ADAPTER * ioc` Pointer to MPT_ADAPTER structure

`char * buffer` Pointer to buffer where IOC summary info should be written

`int * size` Pointer to number of bytes we wrote (set by this routine)

`int len` Offset at which to start writing in buffer

`int showlan` Display LAN stuff?

Description

This routine writes (english readable) ASCII text, which represents a summary of IOC information, to a buffer.

int **`mpt_set_taskmgmt_in_progress_flag`**(MPT_ADAPTER * *ioc*)
set flags associated with task management

Parameters

`MPT_ADAPTER * 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

void **mpt_clear_taskmgmt_in_progress_flag**(MPT_ADAPTER * *ioc*)
clear flags associated with task management

Parameters

MPT_ADAPTER * ioc Pointer to MPT_ADAPTER structure

void **mpt_halt_firmware**(MPT_ADAPTER * *ioc*)
Halts the firmware if it is operational and panic the kernel

Parameters

MPT_ADAPTER * ioc Pointer to MPT_ADAPTER structure

int **mpt_Soft_Hard_ResetHandler**(MPT_ADAPTER * *ioc*, int *sleepFlag*)
Try less expensive reset

Parameters

MPT_ADAPTER * ioc Pointer to MPT_ADAPTER structure

int 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.

int **mpt_HardResetHandler**(MPT_ADAPTER * *ioc*, int *sleepFlag*)
Generic reset handler

Parameters

MPT_ADAPTER * ioc Pointer to MPT_ADAPTER structure

int 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.

const char * **mptscsih_info**(struct Scsi_Host * *SChost*)
Return information about MPT adapter

Parameters

struct Scsi_Host * SChost Pointer to Scsi_Host structure

Description

(linux scsi_host_template.info routine)

Returns pointer to buffer where information was written.

int **mptscsih_qcmd**(struct scsi_cmnd * *SCpnt*)
Primary Fusion MPT SCSI initiator IO start routine.

Parameters

struct scsi_cmnd * 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)

int **`mptscsih_IssueTaskMgmt`**(MPT_SCSI_HOST * *hd*, u8 *type*, u8 *channel*, u8 *id*, u64 *lun*, int *ctx2abort*, ulong *timeout*)
Generic send Task Management function.

Parameters

`MPT_SCSI_HOST * hd` Pointer to MPT_SCSI_HOST structure

`u8 type` Task Management type

`u8 channel` channel number for task management

`u8 id` Logical Target ID for reset (if appropriate)

`u64 lun` Logical Unit for reset (if appropriate)

`int ctx2abort` Context for the task to be aborted (if appropriate)

`ulong timeout` timeout for task management control

Description

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.

int **`mptscsih_abort`**(struct scsi_cmnd * *SCpnt*)
Abort linux scsi_cmnd routine, new_eh variant

Parameters

`struct scsi_cmnd * SCpnt` Pointer to scsi_cmnd structure, IO to be aborted

Description

(linux scsi_host_template.eh_abort_handler routine)

Returns SUCCESS or FAILED.

int **`mptscsih_dev_reset`**(struct scsi_cmnd * *SCpnt*)
Perform a SCSI TARGET_RESET! new_eh variant

Parameters

`struct scsi_cmnd * 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.

int **`mptscsih_bus_reset`**(struct scsi_cmnd * *SCpnt*)
Perform a SCSI BUS_RESET! new_eh variant

Parameters

`struct scsi_cmnd * 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.

int **`mptscsih_host_reset`**(struct scsi_cmnd * *SCpnt*)
Perform a SCSI host adapter RESET (new_eh variant)

Parameters

struct scsi_cmnd * 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.

int **mptscsih_taskmgmt_complete**(MPT_ADAPTER * *ioc*, MPT_FRAME_HDR * *mf*, MPT_FRAME_HDR * *mr*)
Registered with Fusion MPT base driver

Parameters

MPT_ADAPTER * ioc Pointer to MPT_ADAPTER structure

MPT_FRAME_HDR * mf Pointer to SCSI task mgmt request frame

MPT_FRAME_HDR * 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.

struct scsi_cmnd * **mptscsih_get_scsi_lookup**(MPT_ADAPTER * *ioc*, int *i*)
retrieves scmd entry

Parameters

MPT_ADAPTER * ioc Pointer to MPT_ADAPTER structure

int i index into the array

Description

Returns the scsi_cmd pointer

CHAPTER

EIGHT

SOUND DEVICES

snd_printk(*fmt*, *...*)
printk wrapper

Parameters

fmt format string

... variable arguments

Description

Works like *printk()* but prints the file and the line of the caller when configured with CONFIG_SND_VERBOSE_PRINTK.

snd_printd(*fmt*, *...*)
debug printk

Parameters

fmt format string

... variable arguments

Description

Works like *snd_printk()* for debugging purposes. Ignored when CONFIG_SND_DEBUG is not set.

snd_BUG()
give a BUG warning message and stack trace

Parameters

Description

Calls WARN() if CONFIG_SND_DEBUG is set. Ignored when CONFIG_SND_DEBUG is not set.

snd_printd_ratelimit()

Parameters

snd_BUG_ON(*cond*)
debugging check macro

Parameters

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.

snd_printdd(*format*, *...*)
debug printk

Parameters

format format string

... variable arguments

Description

Works like *snd_printk()* for debugging purposes. Ignored when CONFIG_SND_DEBUG_VERBOSE is not set.

int **register_sound_special_device**(const struct file_operations * *fops*, int *unit*, struct *device* * *dev*)
register a special sound node

Parameters

const struct file_operations * fops File operations for the driver

int unit Unit number to allocate

struct device * 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.

int **register_sound_mixer**(const struct file_operations * *fops*, int *dev*)
register a mixer device

Parameters

const struct file_operations * fops File operations for the driver

int 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.

int **register_sound_dsp**(const struct file_operations * *fops*, int *dev*)
register a DSP device

Parameters

const struct file_operations * fops File operations for the driver

int 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.

void **unregister_sound_special**(int *unit*)
unregister a special sound device

Parameters

int 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.

void **unregister_sound_mixer**(int *unit*)
unregister a mixer

Parameters

int 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.

void **unregister_sound_dsp**(int *unit*)
unregister a DSP device

Parameters

int 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.

int **snd_pcm_stream_linked**(struct snd_pcm_substream * *substream*)
Check whether the substream is linked with others

Parameters

struct snd_pcm_substream * substream substream to check

Description

Returns true if the given substream is being linked with others.

snd_pcm_stream_lock_irqsave(*substream*, *flags*)
Lock the PCM stream

Parameters

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()*.

snd_pcm_group_for_each_entry(*s*, *substream*)
iterate over the linked substreams

Parameters

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.

int **snd_pcm_running**(struct snd_pcm_substream * *substream*)
Check whether the substream is in a running state

Parameters

struct snd_pcm_substream * substream substream to check

Description

Returns true if the given substream is in the state RUNNING, or in the state DRAINING for playback.

ssize_t **bytes_to_samples**(struct snd_pcm_runtime * *runtime*, ssize_t *size*)
Unit conversion of the size from bytes to samples

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

ssize_t size size in bytes

snd_pcm_sframes_t **bytes_to_frames**(struct snd_pcm_runtime * *runtime*, ssize_t *size*)
Unit conversion of the size from bytes to frames

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

ssize_t size size in bytes

ssize_t **samples_to_bytes**(struct snd_pcm_runtime * *runtime*, ssize_t *size*)
Unit conversion of the size from samples to bytes

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

ssize_t size size in samples

ssize_t **frames_to_bytes**(struct snd_pcm_runtime * *runtime*, snd_pcm_sframes_t *size*)
Unit conversion of the size from frames to bytes

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

snd_pcm_sframes_t size size in frames

int **frame_aligned**(struct snd_pcm_runtime * *runtime*, ssize_t *bytes*)
Check whether the byte size is aligned to frames

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

ssize_t bytes size in bytes

size_t **snd_pcm_lib_buffer_bytes**(struct snd_pcm_substream * *substream*)
Get the buffer size of the current PCM in bytes

Parameters

struct snd_pcm_substream * substream PCM substream

size_t **snd_pcm_lib_period_bytes**(struct snd_pcm_substream * *substream*)
Get the period size of the current PCM in bytes

Parameters

struct snd_pcm_substream * substream PCM substream

snd_pcm_uframes_t **snd_pcm_playback_avail**(struct snd_pcm_runtime * *runtime*)
Get the available (writable) space for playback

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

Description

Result is between 0 … (boundary - 1)

snd_pcm_uframes_t **snd_pcm_capture_avail**(struct snd_pcm_runtime * *runtime*)
Get the available (readable) space for capture

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

Description

Result is between 0 ... (boundary - 1)

snd_pcm_sframes_t **snd_pcm_playback_hw_avail**(struct snd_pcm_runtime * *runtime*)
Get the queued space for playback

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

snd_pcm_sframes_t **snd_pcm_capture_hw_avail**(struct snd_pcm_runtime * *runtime*)
Get the free space for capture

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

int **snd_pcm_playback_ready**(struct snd_pcm_substream * *substream*)
check whether the playback buffer is available

Parameters

struct snd_pcm_substream * 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.

int **snd_pcm_capture_ready**(struct snd_pcm_substream * *substream*)
check whether the capture buffer is available

Parameters

struct snd_pcm_substream * 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.

int **snd_pcm_playback_data**(struct snd_pcm_substream * *substream*)
check whether any data exists on the playback buffer

Parameters

struct snd_pcm_substream * 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.

int **snd_pcm_playback_empty**(struct snd_pcm_substream * *substream*)
check whether the playback buffer is empty

Parameters

`struct snd_pcm_substream * substream` the pcm substream instance

Description

Checks whether the playback buffer is empty.

Return

Non-zero if empty, or zero if not.

int **`snd_pcm_capture_empty`**(struct snd_pcm_substream * *substream*)
check whether the capture buffer is empty

Parameters

`struct snd_pcm_substream * substream` the pcm substream instance

Description

Checks whether the capture buffer is empty.

Return

Non-zero if empty, or zero if not.

void **`snd_pcm_trigger_done`**(struct snd_pcm_substream * *substream*, struct snd_pcm_substream * *master*)
Mark the master substream

Parameters

`struct snd_pcm_substream * substream` the pcm substream instance

`struct snd_pcm_substream * 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.

unsigned int **`params_channels`**(const struct snd_pcm_hw_params * *p*)
Get the number of channels from the hw params

Parameters

`const struct snd_pcm_hw_params * p` hw params

unsigned int **`params_rate`**(const struct snd_pcm_hw_params * *p*)
Get the sample rate from the hw params

Parameters

`const struct snd_pcm_hw_params * p` hw params

unsigned int **`params_period_size`**(const struct snd_pcm_hw_params * *p*)
Get the period size (in frames) from the hw params

Parameters

`const struct snd_pcm_hw_params * p` hw params

unsigned int **`params_periods`**(const struct snd_pcm_hw_params * *p*)
Get the number of periods from the hw params

Parameters

`const struct snd_pcm_hw_params * p` hw params

unsigned int **`params_buffer_size`**(const struct snd_pcm_hw_params * *p*)
Get the buffer size (in frames) from the hw params

Parameters

const struct snd_pcm_hw_params * p hw params

unsigned int **params_buffer_bytes**(const struct snd_pcm_hw_params * *p*)
Get the buffer size (in bytes) from the hw params

Parameters

const struct snd_pcm_hw_params * p hw params

int **snd_pcm_hw_constraint_single**(struct snd_pcm_runtime * *runtime*, snd_pcm_hw_param_t *var*, unsigned int *val*)
Constrain parameter to a single value

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

snd_pcm_hw_param_t var The hw_params variable to constrain

unsigned int val The value to constrain to

Return

Positive if the value is changed, zero if it's not changed, or a negative error code.

int **snd_pcm_format_cpu_endian**(snd_pcm_format_t *format*)
Check the PCM format is CPU-endian

Parameters

snd_pcm_format_t 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.

void **snd_pcm_set_runtime_buffer**(struct snd_pcm_substream * *substream*, struct snd_dma_buffer * *bufp*)
Set the PCM runtime buffer

Parameters

struct snd_pcm_substream * substream PCM substream to set

struct snd_dma_buffer * 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.

void **snd_pcm_gettime**(struct snd_pcm_runtime * *runtime*, struct timespec * *tv*)
Fill the timespec depending on the timestamp mode

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

struct timespec * tv timespec to fill

int **snd_pcm_lib_alloc_vmalloc_buffer**(struct snd_pcm_substream * *substream*, size_t *size*)
allocate virtual DMA buffer

Parameters

struct snd_pcm_substream * substream the substream to allocate the buffer to

size_t 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.

int **snd_pcm_lib_alloc_vmalloc_32_buffer**(struct snd_pcm_substream * *substream*, size_t *size*)
allocate 32-bit-addressable buffer

Parameters

struct snd_pcm_substream * substream the substream to allocate the buffer to

size_t 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.

dma_addr_t **snd_pcm_sgbuf_get_addr**(struct snd_pcm_substream * *substream*, unsigned int *ofs*)
Get the DMA address at the corresponding offset

Parameters

struct snd_pcm_substream * substream PCM substream

unsigned int ofs byte offset

void * **snd_pcm_sgbuf_get_ptr**(struct snd_pcm_substream * *substream*, unsigned int *ofs*)
Get the virtual address at the corresponding offset

Parameters

struct snd_pcm_substream * substream PCM substream

unsigned int ofs byte offset

unsigned int **snd_pcm_sgbuf_get_chunk_size**(struct snd_pcm_substream * *substream*, unsigned int *ofs*, unsigned int *size*)
Compute the max size that fits within the contig. page from the given size

Parameters

struct snd_pcm_substream * substream PCM substream

unsigned int ofs byte offset

unsigned int size byte size to examine

void **snd_pcm_mmap_data_open**(struct vm_area_struct * *area*)
increase the mmap counter

Parameters

struct vm_area_struct * area VMA

Description

PCM mmap callback should handle this counter properly

void **snd_pcm_mmap_data_close**(struct vm_area_struct * *area*)
decrease the mmap counter

Parameters

struct vm_area_struct * area VMA

Description

PCM mmap callback should handle this counter properly

void **snd_pcm_limit_isa_dma_size**(int *dma*, size_t * *max*)
Get the max size fitting with ISA DMA transfer

Parameters

int dma DMA number

size_t * max pointer to store the max size

const char * **snd_pcm_stream_str**(struct snd_pcm_substream * *substream*)
Get a string naming the direction of a stream

Parameters

struct snd_pcm_substream * substream the pcm substream instance

Return

A string naming the direction of the stream.

struct snd_pcm_substream * **snd_pcm_chmap_substream**(struct snd_pcm_chmap * *info*, unsigned int *idx*)
get the PCM substream assigned to the given chmap info

Parameters

struct snd_pcm_chmap * info chmap information

unsigned int idx the substream number index

u64 **pcm_format_to_bits**(snd_pcm_format_t *pcm_format*)
Strong-typed conversion of pcm_format to bitwise

Parameters

snd_pcm_format_t pcm_format PCM format

const char * **snd_pcm_format_name**(snd_pcm_format_t *format*)
Return a name string for the given PCM format

Parameters

snd_pcm_format_t format PCM format

int **snd_pcm_new_stream**(struct snd_pcm * *pcm*, int *stream*, int *substream_count*)
create a new PCM stream

Parameters

struct snd_pcm * pcm the pcm instance

int stream the stream direction, SNDRV_PCM_STREAM_XXX

int substream_count 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.

int **snd_pcm_new**(struct snd_card * *card*, const char * *id*, int *device*, int *playback_count*, int *capture_count*, struct snd_pcm ** *rpcm*)
create a new PCM instance

Parameters

struct snd_card * card the card instance

const char * id the id string

int device the device index (zero based)

int playback_count the number of substreams for playback

int capture_count the number of substreams for capture

struct snd_pcm ** rpcm 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.

int **snd_pcm_new_internal**(struct snd_card * *card*, const char * *id*, int *device*, int *playback_count*, int *capture_count*, struct snd_pcm ** *rpcm*)
create a new internal PCM instance

Parameters

struct snd_card * card the card instance

const char * id the id string

int device the device index (zero based - shared with normal PCMs)

int playback_count the number of substreams for playback

int capture_count the number of substreams for capture

struct snd_pcm ** rpcm 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 PCMs 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.

int **snd_pcm_notify**(struct *snd_pcm_notify* * *notify*, int *nfree*)
Add/remove the notify list

Parameters

struct snd_pcm_notify * notify PCM notify list

int 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.

int **snd_device_new**(struct snd_card * *card*, enum snd_device_type *type*, void * *device_data*, struct snd_device_ops * *ops*)
create an ALSA device component

Parameters

struct snd_card * card the card instance

enum snd_device_type type the device type, SNDRV_DEV_XXX

void * device_data the data pointer of this device

struct snd_device_ops * ops 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.

void **snd_device_disconnect**(struct snd_card * *card*, void * *device_data*)
disconnect the device

Parameters

struct snd_card * card the card instance

void * 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.

void **snd_device_free**(struct snd_card * *card*, void * *device_data*)
release the device from the card

Parameters

struct snd_card * card the card instance

void * 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.

int **snd_device_register**(struct snd_card * *card*, void * *device_data*)
register the device

Parameters

struct snd_card * card the card instance

void * 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.

int **snd_info_get_line**(struct snd_info_buffer * *buffer*, char * *line*, int *len*)
read one line from the procfs buffer

Parameters

struct snd_info_buffer * buffer the procfs buffer

char * line the buffer to store

int 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.

const char * **snd_info_get_str**(char * *dest*, const char * *src*, int *len*)
parse a string token

Parameters

char * dest the buffer to store the string token

const char * src the original string

int 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.

struct snd_info_entry * **snd_info_create_module_entry**(struct module * *module*, const char * *name*, struct snd_info_entry * *parent*)
create an info entry for the given module

Parameters

struct module * module the module pointer

const char * name the file name

struct snd_info_entry * 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.

struct snd_info_entry * **snd_info_create_card_entry**(struct snd_card * *card*, const char * *name*, struct snd_info_entry * *parent*)
create an info entry for the given card

Parameters

struct snd_card * card the card instance

const char * name the file name

struct snd_info_entry * 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.

void **snd_info_free_entry**(struct snd_info_entry * *entry*)
release the info entry

Parameters

struct snd_info_entry * entry the info entry

Description

Releases the info entry.

int **snd_info_register**(struct snd_info_entry * *entry*)
register the info entry

Parameters

struct snd_info_entry * entry the info entry

Description

Registers the proc info entry.

Return

Zero if successful, or a negative error code on failure.

int **snd_rawmidi_receive**(struct snd_rawmidi_substream * *substream*, const unsigned char * *buffer*, int *count*)
receive the input data from the device

Parameters

struct snd_rawmidi_substream * substream the rawmidi substream

const unsigned char * buffer the buffer pointer

int 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.

int **snd_rawmidi_transmit_empty**(struct snd_rawmidi_substream * *substream*)
check whether the output buffer is empty

Parameters

struct snd_rawmidi_substream * substream the rawmidi substream

Return

1 if the internal output buffer is empty, 0 if not.

int **__snd_rawmidi_transmit_peek**(struct snd_rawmidi_substream * *substream*, unsigned char * *buffer*, int *count*)
copy data from the internal buffer

Parameters

struct snd_rawmidi_substream * substream the rawmidi substream

unsigned char * buffer the buffer pointer

int count data size to transfer

Description

This is a variant of *snd_rawmidi_transmit_peek()* without spinlock.

int **snd_rawmidi_transmit_peek**(struct snd_rawmidi_substream * *substream*, unsigned char * *buffer*, int *count*)
copy data from the internal buffer

Parameters

struct snd_rawmidi_substream * substream the rawmidi substream

unsigned char * buffer the buffer pointer

int 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.

int **__snd_rawmidi_transmit_ack**(struct snd_rawmidi_substream * *substream*, int *count*)
acknowledge the transmission

Parameters

struct snd_rawmidi_substream * substream the rawmidi substream

int count the transferred count

Description

This is a variant of *__snd_rawmidi_transmit_ack()* without spinlock.

int **snd_rawmidi_transmit_ack**(struct snd_rawmidi_substream * *substream*, int *count*)
acknowledge the transmission

Parameters

struct snd_rawmidi_substream * substream the rawmidi substream

int 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.

int **snd_rawmidi_transmit**(struct snd_rawmidi_substream * *substream*, unsigned char * *buffer*, int *count*)
copy from the buffer to the device

Parameters

struct snd_rawmidi_substream * substream the rawmidi substream

unsigned char * buffer the buffer pointer

int 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.

int **snd_rawmidi_new**(struct snd_card * *card*, char * *id*, int *device*, int *output_count*, int *input_count*, struct snd_rawmidi ** *rrawmidi*)
create a rawmidi instance

Parameters

struct snd_card * card the card instance

char * id the id string

int device the device index

int output_count the number of output streams

int input_count the number of input streams

struct snd_rawmidi ** rrawmidi 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.

void **snd_rawmidi_set_ops**(struct snd_rawmidi * *rmidi*, int *stream*, const struct snd_rawmidi_ops * *ops*)
set the rawmidi operators

Parameters

struct snd_rawmidi * rmidi the rawmidi instance

int stream the stream direction, SNDRV_RAWMIDI_STREAM_XXX

const struct snd_rawmidi_ops * ops the operator table

Description

Sets the rawmidi operators for the given stream direction.

void **snd_request_card**(int *card*)
try to load the card module

Parameters

int 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.

void * **snd_lookup_minor_data**(unsigned int *minor*, int *type*)
get user data of a registered device

Parameters

unsigned int minor the minor number

int 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.

int **snd_register_device**(int *type*, struct snd_card * *card*, int *dev*, const struct file_operations * *f_ops*, void * *private_data*, struct device * *device*)
Register the ALSA device file for the card

Parameters

int type the device type, SNDRV_DEVICE_TYPE_XXX

struct snd_card * card the card instance

int dev the device index

const struct file_operations * f_ops the file operations

void * private_data user pointer for f_ops->:c:func:*open()*

struct device * device 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.

int **snd_unregister_device**(struct device * *dev*)
unregister the device on the given card

Parameters

struct device * 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.

int **copy_to_user_fromio**(void __user * *dst*, const volatile void __iomem * *src*, size_t *count*)
copy data from mmio-space to user-space

Parameters

void __user * dst the destination pointer on user-space

const volatile void __iomem * src the source pointer on mmio

size_t 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.

int **copy_from_user_toio**(volatile void __iomem * *dst*, const void __user * *src*, size_t *count*)
copy data from user-space to mmio-space

Parameters

volatile void __iomem * dst the destination pointer on mmio-space

const void __user * src the source pointer on user-space

size_t 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.

int **snd_pcm_lib_preallocate_free_for_all**(struct snd_pcm * *pcm*)
release all pre-allocated buffers on the pcm

Parameters

struct snd_pcm * 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.

int **snd_pcm_lib_preallocate_pages**(struct snd_pcm_substream * *substream*, int *type*, struct *device* * *data*, size_t *size*, size_t *max*)
pre-allocation for the given DMA type

Parameters

struct snd_pcm_substream * substream the pcm substream instance

int type DMA type (SNDRV_DMA_TYPE_*)

struct device * data DMA type dependent data

size_t size the requested pre-allocation size in bytes

size_t max 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.

int **snd_pcm_lib_preallocate_pages_for_all**(struct snd_pcm * *pcm*, int *type*, void * *data*, size_t *size*, size_t *max*)
pre-allocation for continuous memory type (all substreams)

Parameters

struct snd_pcm * pcm the pcm instance

int type DMA type (SNDRV_DMA_TYPE_*)

void * data DMA type dependent data

size_t size the requested pre-allocation size in bytes

size_t 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.

struct page * **snd_pcm_sgbuf_ops_page**(struct snd_pcm_substream * *substream*, unsigned long *offset*)
get the page struct at the given offset

Parameters

struct snd_pcm_substream * substream the pcm substream instance

unsigned long 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.

int **`snd_pcm_lib_malloc_pages`**(struct snd_pcm_substream * *substream*, size_t *size*)
allocate the DMA buffer

Parameters

`struct snd_pcm_substream * substream` the substream to allocate the DMA buffer to

`size_t 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.

int **`snd_pcm_lib_free_pages`**(struct snd_pcm_substream * *substream*)
release the allocated DMA buffer.

Parameters

`struct snd_pcm_substream * 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.

int **`snd_pcm_lib_free_vmalloc_buffer`**(struct snd_pcm_substream * *substream*)
free vmalloc buffer

Parameters

`struct snd_pcm_substream * 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.

struct page * **`snd_pcm_lib_get_vmalloc_page`**(struct snd_pcm_substream * *substream*, unsigned long *offset*)
map vmalloc buffer offset to page struct

Parameters

`struct snd_pcm_substream * substream` the substream with a buffer allocated by *`snd_pcm_lib_alloc_vmalloc_buffer()`*

`unsigned long 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.

void **`snd_device_initialize`**(struct *device* * *dev*, struct snd_card * *card*)
Initialize struct device for sound devices

Parameters

`struct device * dev` device to initialize

`struct snd_card * card` card to assign, optional

int **`snd_card_new`**(struct *device* * *parent*, int *idx*, const char * *xid*, struct module * *module*, int *extra_size*, struct snd_card ** *card_ret*)
create and initialize a soundcard structure

Parameters

`struct device * parent` the parent device object

`int idx` card index (address) [0 … (SNDRV_CARDS-1)]

`const char * xid` card identification (ASCII string)

`struct module * module` top level module for locking

`int extra_size` allocate this extra size after the main soundcard structure

`struct snd_card ** card_ret` 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.

int **`snd_card_disconnect`**(struct snd_card * *card*)
disconnect all APIs from the file-operations (user space)

Parameters

`struct snd_card * 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).

void **`snd_card_disconnect_sync`**(struct snd_card * *card*)
disconnect card and wait until files get closed

Parameters

`struct snd_card * card` card object to disconnect

Description

This calls *`snd_card_disconnect()`* for disconnecting all belonging components and waits until all pending files get closed. It assures that all accesses from user-space finished so that the driver can release its resources gracefully.

int **`snd_card_free_when_closed`**(struct snd_card * *card*)
Disconnect the card, free it later eventually

Parameters

`struct snd_card * 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.

int **snd_card_free**(struct snd_card * *card*)
frees given soundcard structure

Parameters

struct snd_card * 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.

void **snd_card_set_id**(struct snd_card * *card*, const char * *nid*)
set card identification name

Parameters

struct snd_card * card soundcard structure

const char * nid new identification string

Description

This function sets the card identification and checks for name collisions.

int **snd_card_add_dev_attr**(struct snd_card * *card*, const struct attribute_group * *group*)
Append a new sysfs attribute group to card

Parameters

struct snd_card * card card instance

const struct attribute_group * group attribute group to append

int **snd_card_register**(struct snd_card * *card*)
register the soundcard

Parameters

struct snd_card * 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.

int **snd_component_add**(struct snd_card * *card*, const char * *component*)
add a component string

Parameters

struct snd_card * card soundcard structure

const char * 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.

int **snd_card_file_add**(struct snd_card * *card*, struct file * *file*)
add the file to the file list of the card

Parameters

struct snd_card * card soundcard structure

struct file * 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.

int **snd_card_file_remove**(struct snd_card * *card*, struct file * *file*)
remove the file from the file list

Parameters

struct snd_card * card soundcard structure

struct file * 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.

int **snd_power_wait**(struct snd_card * *card*, unsigned int *power_state*)
wait until the power-state is changed.

Parameters

struct snd_card * card soundcard structure

unsigned int power_state expected power state

Description

Waits until the power-state is changed.

Return

Zero if successful, or a negative error code.

void **snd_dma_program**(unsigned long *dma*, unsigned long *addr*, unsigned int *size*, unsigned short *mode*)
program an ISA DMA transfer

Parameters

unsigned long dma the dma number

unsigned long addr the physical address of the buffer

unsigned int size the DMA transfer size

unsigned short mode the DMA transfer mode, DMA_MODE_XXX

Description

Programs an ISA DMA transfer for the given buffer.

void **snd_dma_disable**(unsigned long *dma*)
stop the ISA DMA transfer

Parameters

unsigned long dma the dma number

Description

Stops the ISA DMA transfer.

unsigned int **snd_dma_pointer**(unsigned long *dma*, unsigned int *size*)
return the current pointer to DMA transfer buffer in bytes

Parameters

unsigned long dma the dma number

unsigned int size the dma transfer size

Return

The current pointer in DMA transfer buffer in bytes.

void **snd_ctl_notify**(struct snd_card * *card*, unsigned int *mask*, struct snd_ctl_elem_id * *id*)
Send notification to user-space for a control change

Parameters

struct snd_card * card the card to send notification

unsigned int mask the event mask, SNDRV_CTL_EVENT_*

struct snd_ctl_elem_id * 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.

struct snd_kcontrol * **snd_ctl_new1**(const struct snd_kcontrol_new * *ncontrol*, void * *private_data*)
create a control instance from the template

Parameters

const struct snd_kcontrol_new * ncontrol the initialization record

void * 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.

void **snd_ctl_free_one**(struct snd_kcontrol * *kcontrol*)
release the control instance

Parameters

struct snd_kcontrol * 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.

int **snd_ctl_add**(struct snd_card * *card*, struct snd_kcontrol * *kcontrol*)
add the control instance to the card

Parameters

struct snd_card * card the card instance

struct snd_kcontrol * 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.

int **snd_ctl_replace**(struct snd_card * *card*, struct snd_kcontrol * *kcontrol*, bool *add_on_replace*)
replace the control instance of the card

Parameters

struct snd_card * card the card instance

struct snd_kcontrol * kcontrol the control instance to replace

bool add_on_replace 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.

int **snd_ctl_remove**(struct snd_card * *card*, struct snd_kcontrol * *kcontrol*)
remove the control from the card and release it

Parameters

struct snd_card * card the card instance

struct snd_kcontrol * 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.

int **snd_ctl_remove_id**(struct snd_card * *card*, struct snd_ctl_elem_id * *id*)
remove the control of the given id and release it

Parameters

struct snd_card * card the card instance

struct snd_ctl_elem_id * 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.

int **snd_ctl_activate_id**(struct snd_card * *card*, struct snd_ctl_elem_id * *id*, int *active*)
activate/inactivate the control of the given id

Parameters

struct snd_card * card the card instance

struct snd_ctl_elem_id * id the control id to activate/inactivate

int 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.

int **snd_ctl_rename_id**(struct snd_card * *card*, struct snd_ctl_elem_id * *src_id*, struct snd_ctl_elem_id * *dst_id*)
replace the id of a control on the card

Parameters

struct snd_card * card the card instance

struct snd_ctl_elem_id * src_id the old id

struct snd_ctl_elem_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.

struct snd_kcontrol * **snd_ctl_find_numid**(struct snd_card * *card*, unsigned int *numid*)
find the control instance with the given number-id

Parameters

struct snd_card * card the card instance

unsigned int 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.

struct snd_kcontrol * **snd_ctl_find_id**(struct snd_card * *card*, struct snd_ctl_elem_id * *id*)
find the control instance with the given id

Parameters

struct snd_card * card the card instance

struct snd_ctl_elem_id * 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.

int **snd_ctl_register_ioctl**(snd_kctl_ioctl_func_t *fcn*)
register the device-specific control-ioctls

Parameters

snd_kctl_ioctl_func_t fcn ioctl callback function

Description

called from each device manager like pcm.c, hwdep.c, etc.

int **snd_ctl_register_ioctl_compat**(snd_kctl_ioctl_func_t *fcn*)
register the device-specific 32bit compat control-ioctls

Parameters

snd_kctl_ioctl_func_t fcn ioctl callback function

int **snd_ctl_unregister_ioctl**(snd_kctl_ioctl_func_t *fcn*)
de-register the device-specific control-ioctls

Parameters

snd_kctl_ioctl_func_t fcn ioctl callback function to unregister

int **snd_ctl_unregister_ioctl_compat**(snd_kctl_ioctl_func_t *fcn*)
de-register the device-specific compat 32bit control-ioctls

Parameters

snd_kctl_ioctl_func_t fcn ioctl callback function to unregister

int **snd_ctl_boolean_mono_info**(struct snd_kcontrol * *kcontrol*, struct snd_ctl_elem_info * *uinfo*)
Helper function for a standard boolean info callback with a mono channel

Parameters

struct snd_kcontrol * kcontrol the kcontrol instance

struct snd_ctl_elem_info * 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.

int **snd_ctl_boolean_stereo_info**(struct snd_kcontrol * *kcontrol*, struct snd_ctl_elem_info * *uinfo*)
Helper function for a standard boolean info callback with stereo two channels

Parameters

struct snd_kcontrol * kcontrol the kcontrol instance

struct snd_ctl_elem_info * 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.

int **snd_ctl_enum_info**(struct snd_ctl_elem_info * *info*, unsigned int *channels*, unsigned int *items*, const char *const *names*)
fills the info structure for an enumerated control

Parameters

struct snd_ctl_elem_info * info the structure to be filled

unsigned int channels the number of the control's channels; often one

unsigned int items the number of control values; also the size of **names**

const char *const names 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.

void **snd_pcm_set_ops**(struct snd_pcm * *pcm*, int *direction*, const struct snd_pcm_ops * *ops*)
set the PCM operators

Parameters

struct snd_pcm * pcm the pcm instance

int direction stream direction, SNDRV_PCM_STREAM_XXX

const struct snd_pcm_ops * ops the operator table

Description

Sets the given PCM operators to the pcm instance.

void **snd_pcm_set_sync**(struct snd_pcm_substream * *substream*)
set the PCM sync id

Parameters

struct snd_pcm_substream * substream the pcm substream

Description

Sets the PCM sync identifier for the card.

int **snd_interval_refine**(struct snd_interval * *i*, const struct snd_interval * *v*)
refine the interval value of configurator

Parameters

struct snd_interval * i the interval value to refine

const struct snd_interval * 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.

int **snd_interval_ratnum**(struct snd_interval * *i*, unsigned int *rats_count*, const struct snd_ratnum * *rats*, unsigned int * *nump*, unsigned int * *denp*)
refine the interval value

Parameters

struct snd_interval * i interval to refine

unsigned int rats_count number of ratnum_t

const struct snd_ratnum * rats ratnum_t array

unsigned int * nump pointer to store the resultant numerator

unsigned int * denp pointer to store the resultant denominator

Return

Positive if the value is changed, zero if it's not changed, or a negative error code.

int **snd_interval_list**(struct snd_interval * *i*, unsigned int *count*, const unsigned int * *list*, unsigned int *mask*)
refine the interval value from the list

Parameters

struct snd_interval * i the interval value to refine

unsigned int count the number of elements in the list

const unsigned int * list the value list

unsigned int 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.

int **snd_interval_ranges**(struct snd_interval * *i*, unsigned int *count*, const struct snd_interval * *ranges*, unsigned int *mask*)
refine the interval value from the list of ranges

Parameters

struct snd_interval * i the interval value to refine

unsigned int count the number of elements in the list of ranges

const struct snd_interval * ranges the ranges list

unsigned int mask 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.

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*, ...)
add the hw-constraint rule

Parameters

struct snd_pcm_runtime * runtime the pcm runtime instance

unsigned int cond condition bits

int var the variable to evaluate

snd_pcm_hw_rule_func_t func the evaluation function

void * private the private data pointer passed to function

int dep the dependent variables

... variable arguments

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_hw_constraint_mask64**(struct snd_pcm_runtime * *runtime*, snd_pcm_hw_param_t *var*, u_int64_t *mask*)
apply the given bitmap mask constraint

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

snd_pcm_hw_param_t var hw_params variable to apply the mask

u_int64_t 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.

int **snd_pcm_hw_constraint_integer**(struct snd_pcm_runtime * *runtime*, snd_pcm_hw_param_t *var*)
apply an integer constraint to an interval

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

snd_pcm_hw_param_t 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.

int **snd_pcm_hw_constraint_minmax**(struct snd_pcm_runtime * *runtime*, snd_pcm_hw_param_t *var*, unsigned int *min*, unsigned int *max*)
apply a min/max range constraint to an interval

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

snd_pcm_hw_param_t var hw_params variable to apply the range

unsigned int min the minimal value

unsigned int max 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.

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*)
apply a list of constraints to a parameter

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

unsigned int cond condition bits

snd_pcm_hw_param_t var hw_params variable to apply the list constraint

const struct snd_pcm_hw_constraint_list * l list

Description

Apply the list of constraints to an interval parameter.

Return

Zero if successful, or a negative error code on failure.

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*)
apply list of range constraints to a parameter

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

unsigned int cond condition bits

snd_pcm_hw_param_t var hw_params variable to apply the list of range constraints

const struct snd_pcm_hw_constraint_ranges * r ranges

Description

Apply the list of range constraints to an interval parameter.

Return

Zero if successful, or a negative error code on failure.

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*)
apply ratnums constraint to a parameter

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

unsigned int cond condition bits

snd_pcm_hw_param_t var hw_params variable to apply the ratnums constraint

const struct snd_pcm_hw_constraint_ratnums * r struct snd_ratnums constriants

Return

Zero if successful, or a negative error code on failure.

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*)
apply ratdens constraint to a parameter

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

unsigned int cond condition bits

snd_pcm_hw_param_t var hw_params variable to apply the ratdens constraint

const struct snd_pcm_hw_constraint_ratdens * r struct snd_ratdens constriants

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_hw_constraint_msbits**(struct snd_pcm_runtime * *runtime*, unsigned int *cond*, unsigned int *width*, unsigned int *msbits*)
add a hw constraint msbits rule

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

unsigned int cond condition bits

unsigned int width sample bits width

unsigned int 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.

int **snd_pcm_hw_constraint_step**(struct snd_pcm_runtime * *runtime*, unsigned int *cond*, snd_pcm_hw_param_t *var*, unsigned long *step*)
add a hw constraint step rule

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

unsigned int cond condition bits

snd_pcm_hw_param_t var hw_params variable to apply the step constraint

unsigned long step step size

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_hw_constraint_pow2**(struct snd_pcm_runtime * *runtime*, unsigned int *cond*, snd_pcm_hw_param_t *var*)
add a hw constraint power-of-2 rule

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

unsigned int cond condition bits

snd_pcm_hw_param_t var hw_params variable to apply the power-of-2 constraint

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_hw_rule_noresample**(struct snd_pcm_runtime * *runtime*, unsigned int *base_rate*)
add a rule to allow disabling hw resampling

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

unsigned int base_rate the rate at which the hardware does not resample

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_hw_param_value**(const struct snd_pcm_hw_params * *params*, snd_pcm_hw_param_t *var*, int * *dir*)
return **params** field **var** value

Parameters

const struct snd_pcm_hw_params * params the hw_params instance

snd_pcm_hw_param_t var parameter to retrieve

int * 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.

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*)
refine config space and return minimum value

Parameters

struct snd_pcm_substream * pcm PCM instance

struct snd_pcm_hw_params * params the hw_params instance

snd_pcm_hw_param_t var parameter to retrieve

int * dir 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.

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*)
refine config space and return maximum value

Parameters

struct snd_pcm_substream * pcm PCM instance

struct snd_pcm_hw_params * params the hw_params instance

snd_pcm_hw_param_t var parameter to retrieve

int * dir 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.

int **snd_pcm_lib_ioctl**(struct snd_pcm_substream * *substream*, unsigned int *cmd*, void * *arg*)
a generic PCM ioctl callback

Parameters

struct snd_pcm_substream * substream the pcm substream instance

unsigned int cmd ioctl command

void * 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.

void **snd_pcm_period_elapsed**(struct snd_pcm_substream * *substream*)
update the pcm status for the next period

Parameters

`struct snd_pcm_substream * 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.

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*)
create channel-mapping control elements

Parameters

`struct snd_pcm * pcm` the assigned PCM instance

`int stream` stream direction

`const struct snd_pcm_chmap_elem * chmap` channel map elements (for query)

`int max_channels` the max number of channels for the stream

`unsigned long private_value` the value passed to each kcontrol's private_value field

`struct snd_pcm_chmap ** info_ret` store struct snd_pcm_chmap 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.

int **`snd_hwdep_new`**(struct snd_card * *card*, char * *id*, int *device*, struct snd_hwdep ** *rhwdep*)
create a new hwdep instance

Parameters

`struct snd_card * card` the card instance

`char * id` the id string

`int device` the device index (zero-based)

`struct snd_hwdep ** 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.

void **`snd_pcm_stream_lock`**(struct snd_pcm_substream * *substream*)
Lock the PCM stream

Parameters

`struct snd_pcm_substream * 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.

void **`snd_pcm_stream_unlock`**(struct snd_pcm_substream * *substream*)
Unlock the PCM stream

Parameters

struct snd_pcm_substream * substream PCM substream

Description

This unlocks the PCM stream that has been locked via *snd_pcm_stream_lock()*.

void **snd_pcm_stream_lock_irq**(struct snd_pcm_substream * *substream*)
Lock the PCM stream

Parameters

struct snd_pcm_substream * 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()*.

void **snd_pcm_stream_unlock_irq**(struct snd_pcm_substream * *substream*)
Unlock the PCM stream

Parameters

struct snd_pcm_substream * substream PCM substream

Description

This is a counter-part of *snd_pcm_stream_lock_irq()*.

void **snd_pcm_stream_unlock_irqrestore**(struct snd_pcm_substream * *substream*, unsigned long *flags*)
Unlock the PCM stream

Parameters

struct snd_pcm_substream * substream PCM substream

unsigned long flags irq flags

Description

This is a counter-part of *snd_pcm_stream_lock_irqsave()*.

int **snd_pcm_stop**(struct snd_pcm_substream * *substream*, snd_pcm_state_t *state*)
try to stop all running streams in the substream group

Parameters

struct snd_pcm_substream * substream the PCM substream instance

snd_pcm_state_t 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.

int **snd_pcm_stop_xrun**(struct snd_pcm_substream * *substream*)
stop the running streams as XRUN

Parameters

struct snd_pcm_substream * 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.

int **snd_pcm_suspend**(struct snd_pcm_substream * *substream*)
trigger SUSPEND to all linked streams

Parameters

struct snd_pcm_substream * 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.

int **snd_pcm_suspend_all**(struct snd_pcm * *pcm*)
trigger SUSPEND to all substreams in the given pcm

Parameters

struct snd_pcm * 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.

int **snd_pcm_kernel_ioctl**(struct snd_pcm_substream * *substream*, unsigned int *cmd*, void * *arg*)
Execute PCM ioctl in the kernel-space

Parameters

struct snd_pcm_substream * substream PCM substream

unsigned int cmd IOCTL cmd

void * arg IOCTL argument

Description

The function is provided primarily for OSS layer and USB gadget drivers, and it allows only the limited set of ioctls (hw_params, sw_params, prepare, start, drain, drop, forward).

int **snd_pcm_lib_default_mmap**(struct snd_pcm_substream * *substream*, struct vm_area_struct * *area*)
Default PCM data mmap function

Parameters

struct snd_pcm_substream * substream PCM substream

struct vm_area_struct * area VMA

Description

This is the default mmap handler for PCM data. When mmap pcm_ops is NULL, this function is invoked implicitly.

int **snd_pcm_lib_mmap_iomem**(struct snd_pcm_substream * *substream*, struct vm_area_struct * *area*)
Default PCM data mmap function for I/O mem

Parameters

struct snd_pcm_substream * substream PCM substream

struct vm_area_struct * 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.

void * **snd_malloc_pages**(size_t *size*, gfp_t *gfp_flags*)
    allocate pages with the given size

Parameters

size_t size the size to allocate in bytes

gfp_t 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.

void **snd_free_pages**(void * *ptr*, size_t *size*)
    release the pages

Parameters

void * ptr the buffer pointer to release

size_t size the allocated buffer size

Description

Releases the buffer allocated via *snd_malloc_pages()*.

int **snd_dma_alloc_pages**(int *type*, struct *device* * *device*, size_t *size*, struct snd_dma_buffer * *dmab*)
    allocate the buffer area according to the given type

Parameters

int type the DMA buffer type

struct device * device the device pointer

size_t size the buffer size to allocate

struct snd_dma_buffer * 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.

int **snd_dma_alloc_pages_fallback**(int *type*, struct *device* * *device*, size_t *size*, struct snd_dma_buffer * *dmab*)
    allocate the buffer area according to the given type with fallback

Parameters

int type the DMA buffer type

struct device * device the device pointer

size_t size the buffer size to allocate

struct snd_dma_buffer * 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.

void **`snd_dma_free_pages`**(struct snd_dma_buffer * *dmab*)
release the allocated buffer

Parameters

`struct snd_dma_buffer * dmab` the buffer allocation record to release

Description

Releases the allocated buffer via *`snd_dma_alloc_pages()`*.

CHAPTER

NINE

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

int **`register_framebuffer`**(struct fb_info * *fb_info*)
    registers a frame buffer device

Parameters

`struct fb_info * fb_info` frame buffer info structure

Description

Registers a frame buffer device **fb_info**.

Returns negative errno on error, or zero for success.

int **`unregister_framebuffer`**(struct fb_info * *fb_info*)
    releases a frame buffer device

Parameters

`struct fb_info * 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.

void **fb_set_suspend**(struct fb_info * *info*, int *state*)
low level driver signals suspend

Parameters

struct fb_info * info framebuffer affected

int 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

void **fb_dealloc_cmap**(struct fb_cmap * *cmap*)
deallocate a colormap

Parameters

struct fb_cmap * cmap frame buffer colormap structure

Description

Deallocates a colormap that was previously allocated with `fb_alloc_cmap()`.

int **fb_copy_cmap**(const struct fb_cmap * *from*, struct fb_cmap * *to*)
copy a colormap

Parameters

const struct fb_cmap * from frame buffer colormap structure

struct fb_cmap * to frame buffer colormap structure

Description

Copy contents of colormap from **from** to **to**.

int **fb_set_cmap**(struct fb_cmap * *cmap*, struct fb_info * *info*)
set the colormap

Parameters

struct fb_cmap * cmap frame buffer colormap structure

struct fb_info * 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.

const struct fb_cmap * **fb_default_cmap**(int *len*)
get default colormap

Parameters

int 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.

void **`fb_invert_cmaps`**(void)
invert all defaults colormaps

Parameters

`void` no arguments

Description

Invert all default colormaps.

Frame Buffer Video Mode Database

int **`fb_try_mode`**(struct fb_var_screeninfo * *var*, struct fb_info * *info*, const struct fb_videomode * *mode*, unsigned int *bpp*)
test a video mode

Parameters

`struct fb_var_screeninfo * var` frame buffer user defined part of display

`struct fb_info * info` frame buffer info structure

`const struct fb_videomode * mode` frame buffer video mode structure

`unsigned int bpp` color depth in bits per pixel

Description

Tries a video mode to test it's validity for device **info**.

Returns 1 on success.

void **`fb_delete_videomode`**(const struct fb_videomode * *mode*, struct list_head * *head*)
removed videomode entry from modelist

Parameters

`const struct fb_videomode * mode` videomode to remove

`struct list_head * head` struct list_head of modelist

NOTES

Will remove all matching mode entries

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*)
finds a valid video mode

Parameters

`struct fb_var_screeninfo * var` frame buffer user defined part of display

`struct fb_info * info` frame buffer info structure

`const char * mode_option` string video mode to find

`const struct fb_videomode * db` video mode database

`unsigned int dbsize` size of **db**

`const struct fb_videomode * default_mode` default video mode to fall back to

`unsigned int default_bpp` default color depth in bits per pixel

Description

Finds a suitable video mode, starting with the specified mode in **mode_option** with fallback to **default_mode**. If **default_mode** fails, all modes in the video mode database will be tried.

Valid mode specifiers for **mode_option**:

<xres>x<yres>[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-8**60m** - 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.

void **fb_var_to_videomode**(struct fb_videomode * *mode*, const struct fb_var_screeninfo * *var*)
convert fb_var_screeninfo to fb_videomode

Parameters

struct fb_videomode * mode pointer to struct fb_videomode

const struct fb_var_screeninfo * var pointer to struct fb_var_screeninfo

void **fb_videomode_to_var**(struct fb_var_screeninfo * *var*, const struct fb_videomode * *mode*)
convert fb_videomode to fb_var_screeninfo

Parameters

struct fb_var_screeninfo * var pointer to struct fb_var_screeninfo

const struct fb_videomode * mode pointer to struct fb_videomode

int **fb_mode_is_equal**(const struct fb_videomode * *mode1*, const struct fb_videomode * *mode2*)
compare 2 videomodes

Parameters

const struct fb_videomode * mode1 first videomode

const struct fb_videomode * mode2 second videomode

Return

1 if equal, 0 if not

const struct fb_videomode * **fb_find_best_mode**(const struct fb_var_screeninfo * *var*, struct list_head * *head*)
find best matching videomode

Parameters

const struct fb_var_screeninfo * var pointer to struct fb_var_screeninfo

struct list_head * head pointer to struct list_head of modelist

Return

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

const struct fb_videomode * **`fb_find_nearest_mode`**(const struct fb_videomode * *mode*, struct list_head * *head*)

find closest videomode

Parameters

`const struct fb_videomode * mode` pointer to struct fb_videomode

`struct list_head * 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.

const struct fb_videomode * **`fb_match_mode`**(const struct fb_var_screeninfo * *var*, struct list_head * *head*)

find a videomode which exactly matches the timings in var

Parameters

`const struct fb_var_screeninfo * var` pointer to struct fb_var_screeninfo

`struct list_head * head` pointer to struct list_head of modelist

Return

struct fb_videomode, NULL if none found

int **`fb_add_videomode`**(const struct fb_videomode * *mode*, struct list_head * *head*)

adds videomode entry to modelist

Parameters

`const struct fb_videomode * mode` videomode to add

`struct list_head * head` struct list_head of modelist

NOTES

Will only add unmatched mode entries

void **`fb_destroy_modelist`**(struct list_head * *head*)

destroy modelist

Parameters

`struct list_head * head` struct list_head of modelist

void **`fb_videomode_to_modelist`**(const struct fb_videomode * *modedb*, int *num*, struct list_head * *head*)

convert mode array to mode list

Parameters

`const struct fb_videomode * modedb` array of struct fb_videomode

`int num` number of entries in array

`struct list_head * head` struct list_head of modelist

Frame Buffer Macintosh Video Mode Database

int **`mac_vmode_to_var`**(int *vmode*, int *cmode*, struct fb_var_screeninfo * *var*)

converts vmode/cmode pair to var structure

Parameters

`int vmode` MacOS video mode

`int cmode` MacOS color mode

`struct fb_var_screeninfo * 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.

int **`mac_map_monitor_sense`**(int *sense*)
Convert monitor sense to vmode

Parameters

`int sense` Macintosh monitor sense number

Description

Converts a Macintosh monitor sense number to a MacOS vmode number.

Returns MacOS vmode video mode number.

int **`mac_find_mode`**(struct fb_var_screeninfo * *var*, struct fb_info * *info*, const char * *mode_option*, unsigned int *default_bpp*)
find a video mode

Parameters

`struct fb_var_screeninfo * var` frame buffer user defined part of display

`struct fb_info * info` frame buffer info structure

`const char * mode_option` video mode name (see mac_modedb[])

`unsigned int default_bpp` default color depth in bits per pixel

Description

Finds a suitable video mode. Tries to set mode specified by **mode_option**. If the name of the wanted mode begins with 'mac', the Mac video mode database will be used, otherwise it will fall back to the standard video mode database.

Note

Function marked as __init and can only be used during system boot.

Returns error code from fb_find_mode (see fb_find_mode function).

Frame Buffer Fonts

Refer to the file lib/fonts/fonts.c for more information.

CHAPTER

TEN

VOLTAGE AND CURRENT REGULATOR API

Author Liam Girdwood

Author Mark Brown

Introduction

This framework is designed to provide a standard kernel interface to control voltage and current regulators.

The intention is to allow systems to dynamically control regulator power output in order to save power and prolong battery life. This applies to both voltage regulators (where voltage output is controllable) and current sinks (where current limit is controllable).

Note that additional (and currently more complete) documentation is available in the Linux kernel source under `Documentation/power/regulator`.

Glossary

The regulator API uses a number of terms which may not be familiar:

Regulator

Electronic device that supplies power to other devices. Most regulators can enable and disable their output and some can also control their output voltage or current.

Consumer

Electronic device which consumes power provided by a regulator. These may either be static, requiring only a fixed supply, or dynamic, requiring active management of the regulator at runtime.

Power Domain

The electronic circuit supplied by a given regulator, including the regulator and all consumer devices. The configuration of the regulator is shared between all the components in the circuit.

Power Management Integrated Circuit (PMIC)

An IC which contains numerous regulators and often also other subsystems. In an embedded system the primary PMIC is often equivalent to a combination of the PSU and southbridge in a desktop system.

Consumer driver interface

This offers a similar API to the kernel clock framework. Consumer drivers use *get* and *put* operations to acquire and release regulators. Functions are provided to *enable* and *disable* the regulator and to get and set the runtime parameters of the regulator.

When requesting regulators consumers use symbolic names for their supplies, such as “Vcc”, which are mapped into actual regulator devices by the machine interface.

A stub version of this API is provided when the regulator framework is not in use in order to minimise the need to use ifdefs.

Enabling and disabling

The regulator API provides reference counted enabling and disabling of regulators. Consumer devices use the *`regulator_enable()`* and *`regulator_disable()`* functions to enable and disable regulators. Calls to the two functions must be balanced.

Note that since multiple consumers may be using a regulator and machine constraints may not allow the regulator to be disabled there is no guarantee that calling *`regulator_disable()`* will actually cause the supply provided by the regulator to be disabled. Consumer drivers should assume that the regulator may be enabled at all times.

Configuration

Some consumer devices may need to be able to dynamically configure their supplies. For example, MMC drivers may need to select the correct operating voltage for their cards. This may be done while the regulator is enabled or disabled.

The *`regulator_set_voltage()`* and *`regulator_set_current_limit()`* functions provide the primary interface for this. Both take ranges of voltages and currents, supporting drivers that do not require a specific value (eg, CPU frequency scaling normally permits the CPU to use a wider range of supply voltages at lower frequencies but does not require that the supply voltage be lowered). Where an exact value is required both minimum and maximum values should be identical.

Callbacks

Callbacks may also be registered for events such as regulation failures.

Regulator driver interface

Drivers for regulator chips register the regulators with the regulator core, providing operations structures to the core. A notifier interface allows error conditions to be reported to the core.

Registration should be triggered by explicit setup done by the platform, supplying a struct *`regulator_init_data`* for the regulator containing constraint and supply information.

Machine interface

This interface provides a way to define how regulators are connected to consumers on a given system and what the valid operating parameters are for the system.

Supplies

Regulator supplies are specified using struct *`regulator_consumer_supply`*. This is done at driver registration time as part of the machine constraints.

Constraints

As well as defining the connections the machine interface also provides constraints defining the operations that clients are allowed to perform and the parameters that may be set. This is required since generally regulator devices will offer more flexibility than it is safe to use on a given system, for example supporting higher supply voltages than the consumers are rated for.

This is done at driver registration time' by providing a struct *regulation_constraints*.

The constraints may also specify an initial configuration for the regulator in the constraints, which is particularly useful for use with static consumers.

API reference

Due to limitations of the kernel documentation framework and the existing layout of the source code the entire regulator API is documented here.

struct **pre_voltage_change_data**
Data sent with PRE_VOLTAGE_CHANGE event

Definition

```
struct pre_voltage_change_data {
  unsigned long old_uV;
  unsigned long min_uV;
  unsigned long max_uV;
};
```

Members

old_uV Current voltage before change.

min_uV Min voltage we'll change to.

max_uV Max voltage we'll change to.

struct **regulator_bulk_data**
Data used for bulk regulator operations.

Definition

```
struct regulator_bulk_data {
  const char *supply;
  struct regulator *consumer;
};
```

Members

supply The name of the supply. Initialised by the user before using the bulk regulator APIs.

consumer The regulator consumer for the supply. This will be managed by the bulk API.

Description

The regulator APIs provide a series of `regulator_bulk_()` API calls as a convenience to consumers which require multiple supplies. This structure is used to manage data for these calls.

struct **regulator_state**
regulator state during low power system states

Definition

```
struct regulator_state {
  int uV;
  int min_uV;
  int max_uV;
  unsigned int mode;
  int enabled;
  bool changeable;
};
```

Members

uV Default operating voltage during suspend, it can be adjusted among <min_uV, max_uV>.

min_uV Minimum suspend voltage may be set.

max_uV Maximum suspend voltage may be set.

mode Operating mode during suspend.

enabled operations during suspend. - DO_NOTHING_IN_SUSPEND - DISABLE_IN_SUSPEND - ENABLE_IN_SUSPEND

changeable Is this state can be switched between enabled/disabled,

Description

This describes a regulators state during a system wide low power state. One of enabled or disabled must be set for the configuration to be applied.

struct **regulation_constraints**
regulator operating constraints.

Definition

```
struct regulation_constraints {
  const char *name;
  int min_uV;
  int max_uV;
  int uV_offset;
  int min_uA;
  int max_uA;
  int ilim_uA;
  int system_load;
  unsigned int valid_modes_mask;
  unsigned int valid_ops_mask;
  int input_uV;
  struct regulator_state state_disk;
  struct regulator_state state_mem;
  struct regulator_state state_standby;
  suspend_state_t initial_state;
  unsigned int initial_mode;
  unsigned int ramp_delay;
  unsigned int settling_time;
  unsigned int settling_time_up;
  unsigned int settling_time_down;
  unsigned int enable_time;
  unsigned int active_discharge;
  unsigned always_on:1;
  unsigned boot_on:1;
  unsigned apply_uV:1;
  unsigned ramp_disable:1;
  unsigned soft_start:1;
  unsigned pull_down:1;
  unsigned over_current_protection:1;
};
```

Members

name Descriptive name for the constraints, used for display purposes.

min_uV Smallest voltage consumers may set.

max_uV Largest voltage consumers may set.

uV_offset Offset applied to voltages from consumer to compensate for voltage drops.

min_uA Smallest current consumers may set.

max_uA Largest current consumers may set.

ilim_uA Maximum input current.

system_load Load that isn't captured by any consumer requests.

valid_modes_mask Mask of modes which may be configured by consumers.

valid_ops_mask Operations which may be performed by consumers.

input_uV Input voltage for regulator when supplied by another regulator.

state_disk State for regulator when system is suspended in disk mode.

state_mem State for regulator when system is suspended in mem mode.

state_standby State for regulator when system is suspended in standby mode.

initial_state Suspend state to set by default.

initial_mode Mode to set at startup.

ramp_delay Time to settle down after voltage change (unit: uV/us)

settling_time Time to settle down after voltage change when voltage change is non-linear (unit: microseconds).

settling_time_up Time to settle down after voltage increase when voltage change is non-linear (unit: microseconds).

settling_time_down Time to settle down after voltage decrease when voltage change is non-linear (unit: microseconds).

enable_time Turn-on time of the rails (unit: microseconds)

active_discharge Enable/disable active discharge. The enum regulator_active_discharge values are used for initialisation.

always_on Set if the regulator should never be disabled.

boot_on Set if the regulator is enabled when the system is initially started. If the regulator is not enabled by the hardware or bootloader then it will be enabled when the constraints are applied.

apply_uV Apply the voltage constraint when initialising.

ramp_disable Disable ramp delay when initialising or when setting voltage.

soft_start Enable soft start so that voltage ramps slowly.

pull_down Enable pull down when regulator is disabled.

over_current_protection Auto disable on over current event.

Description

This struct describes regulator and board/machine specific constraints.

struct **regulator_consumer_supply**
supply -> device mapping

Definition

```
struct regulator_consumer_supply {
  const char *dev_name;
  const char *supply;
};
```

Members

dev_name Result of dev_name() for the consumer.

supply Name for the supply.

Description

This maps a supply name to a device. Use of dev_name allows support for buses which make struct device available late such as I2C.

struct **regulator_init_data**
regulator platform initialisation data.

Definition

```
struct regulator_init_data {
  const char *supply_regulator;
  struct regulation_constraints constraints;
  int num_consumer_supplies;
  struct regulator_consumer_supply *consumer_supplies;
  int (*regulator_init)(void *driver_data);
  void *driver_data;
};
```

Members

supply_regulator Parent regulator. Specified using the regulator name as it appears in the name field in sysfs, which can be explicitly set using the constraints field 'name'.

constraints Constraints. These must be specified for the regulator to be usable.

num_consumer_supplies Number of consumer device supplies.

consumer_supplies Consumer device supply configuration.

regulator_init Callback invoked when the regulator has been registered.

driver_data Data passed to regulator_init.

Description

Initialisation constraints, our supply and consumers supplies.

struct **regulator_linear_range**
specify linear voltage ranges

Definition

```
struct regulator_linear_range {
  unsigned int min_uV;
  unsigned int min_sel;
  unsigned int max_sel;
  unsigned int uV_step;
};
```

Members

min_uV Lowest voltage in range

min_sel Lowest selector for range

max_sel Highest selector for range

uV_step Step size

Description

Specify a range of voltages for regulator_map_linar_range() and regulator_list_linear_range().

struct **regulator_ops**
regulator operations.

Definition

```
struct regulator_ops {
  int (*list_voltage) (struct regulator_dev *, unsigned selector);
  int (*set_voltage) (struct regulator_dev *, int min_uV, int max_uV, unsigned *selector);
  int (*map_voltage)(struct regulator_dev *, int min_uV, int max_uV);
  int (*set_voltage_sel) (struct regulator_dev *, unsigned selector);
  int (*get_voltage) (struct regulator_dev *);
  int (*get_voltage_sel) (struct regulator_dev *);
  int (*set_current_limit) (struct regulator_dev *, int min_uA, int max_uA);
  int (*get_current_limit) (struct regulator_dev *);
  int (*set_input_current_limit) (struct regulator_dev *, int lim_uA);
  int (*set_over_current_protection) (struct regulator_dev *);
  int (*set_active_discharge) (struct regulator_dev *, bool enable);
  int (*enable) (struct regulator_dev *);
  int (*disable) (struct regulator_dev *);
  int (*is_enabled) (struct regulator_dev *);
  int (*set_mode) (struct regulator_dev *, unsigned int mode);
  unsigned int (*get_mode) (struct regulator_dev *);
  int (*get_error_flags)(struct regulator_dev *, unsigned int *flags);
  int (*enable_time) (struct regulator_dev *);
  int (*set_ramp_delay) (struct regulator_dev *, int ramp_delay);
  int (*set_voltage_time) (struct regulator_dev *, int old_uV, int new_uV);
  int (*set_voltage_time_sel) (struct regulator_dev *,unsigned int old_selector, unsigned int new_select
  int (*set_soft_start) (struct regulator_dev *);
  int (*get_status)(struct regulator_dev *);
  unsigned int (*get_optimum_mode) (struct regulator_dev *, int input_uV, int output_uV, int load_uA);
  int (*set_load)(struct regulator_dev *, int load_uA);
  int (*set_bypass)(struct regulator_dev *dev, bool enable);
  int (*get_bypass)(struct regulator_dev *dev, bool *enable);
  int (*set_suspend_voltage) (struct regulator_dev *, int uV);
  int (*set_suspend_enable) (struct regulator_dev *);
  int (*set_suspend_disable) (struct regulator_dev *);
  int (*set_suspend_mode) (struct regulator_dev *, unsigned int mode);
  int (*resume_early)(struct regulator_dev *rdev);
  int (*set_pull_down) (struct regulator_dev *);
};
```

Members

list_voltage Return one of the supported voltages, in microvolts; zero if the selector indicates a voltage that is unusable on this system; or negative errno. Selectors range from zero to one less than regulator_desc.n_voltages. Voltages may be reported in any order.

set_voltage Set the voltage for the regulator within the range specified. The driver should select the voltage closest to min_uV.

map_voltage Convert a voltage into a selector

set_voltage_sel Set the voltage for the regulator using the specified selector.

get_voltage Return the currently configured voltage for the regulator.

get_voltage_sel Return the currently configured voltage selector for the regulator.

set_current_limit Configure a limit for a current-limited regulator. The driver should select the current closest to max_uA.

get_current_limit Get the configured limit for a current-limited regulator.

set_input_current_limit Configure an input limit.

set_over_current_protection Support capability of automatically shutting down when detecting an over current event.

set_active_discharge Set active discharge enable/disable of regulators.

enable Configure the regulator as enabled.

disable Configure the regulator as disabled.

is_enabled Return 1 if the regulator is enabled, 0 if not. May also return negative errno.

set_mode Set the configured operating mode for the regulator.

get_mode Get the configured operating mode for the regulator.

get_error_flags Get the current error(s) for the regulator.

enable_time Time taken for the regulator voltage output voltage to stabilise after being enabled, in microseconds.

set_ramp_delay Set the ramp delay for the regulator. The driver should select ramp delay equal to or less than(closest) ramp_delay.

set_voltage_time Time taken for the regulator voltage output voltage to stabilise after being set to a new value, in microseconds. The function receives the from and to voltage as input, it should return the worst case.

set_voltage_time_sel Time taken for the regulator voltage output voltage to stabilise after being set to a new value, in microseconds. The function receives the from and to voltage selector as input, it should return the worst case.

set_soft_start Enable soft start for the regulator.

get_status Return actual (not as-configured) status of regulator, as a REGULATOR_STATUS value (or negative errno)

get_optimum_mode Get the most efficient operating mode for the regulator when running with the specified parameters.

set_load Set the load for the regulator.

set_bypass Set the regulator in bypass mode.

get_bypass Get the regulator bypass mode state.

set_suspend_voltage Set the voltage for the regulator when the system is suspended.

set_suspend_enable Mark the regulator as enabled when the system is suspended.

set_suspend_disable Mark the regulator as disabled when the system is suspended.

set_suspend_mode Set the operating mode for the regulator when the system is suspended.

set_pull_down Configure the regulator to pull down when the regulator is disabled.

Description

This struct describes regulator operations which can be implemented by regulator chip drivers.

struct **regulator_desc**
Static regulator descriptor

Definition

```
struct regulator_desc {
  const char *name;
  const char *supply_name;
  const char *of_match;
  const char *regulators_node;
  int (*of_parse_cb)(struct device_node *,const struct regulator_desc *, struct regulator_config *);
  int id;
  unsigned int continuous_voltage_range:1;
  unsigned n_voltages;
  const struct regulator_ops *ops;
  int irq;
  enum regulator_type type;
  struct module *owner;
  unsigned int min_uV;
  unsigned int uV_step;
  unsigned int linear_min_sel;
  int fixed_uV;
  unsigned int ramp_delay;
  int min_dropout_uV;
  const struct regulator_linear_range *linear_ranges;
  int n_linear_ranges;
  const unsigned int *volt_table;
  unsigned int vsel_reg;
  unsigned int vsel_mask;
  unsigned int csel_reg;
  unsigned int csel_mask;
  unsigned int apply_reg;
  unsigned int apply_bit;
  unsigned int enable_reg;
  unsigned int enable_mask;
  unsigned int enable_val;
  unsigned int disable_val;
  bool enable_is_inverted;
  unsigned int bypass_reg;
  unsigned int bypass_mask;
  unsigned int bypass_val_on;
  unsigned int bypass_val_off;
  unsigned int active_discharge_on;
  unsigned int active_discharge_off;
  unsigned int active_discharge_mask;
  unsigned int active_discharge_reg;
  unsigned int soft_start_reg;
  unsigned int soft_start_mask;
  unsigned int soft_start_val_on;
  unsigned int pull_down_reg;
  unsigned int pull_down_mask;
  unsigned int pull_down_val_on;
  unsigned int enable_time;
  unsigned int off_on_delay;
  unsigned int (*of_map_mode)(unsigned int mode);
};
```

Members

name Identifying name for the regulator.

supply_name Identifying the regulator supply

of_match Name used to identify regulator in DT.

regulators_node Name of node containing regulator definitions in DT.

of_parse_cb Optional callback called only if of_match is present. Will be called for each regulator parsed from DT, during init_data parsing. The regulator_config passed as argument to the callback will be a copy of config passed to regulator_register, valid only for this particular call. Callback may freely change the config but it cannot store it for later usage. Callback should return 0 on success or negative ERRNO indicating failure.

id Numerical identifier for the regulator.

continuous_voltage_range Indicates if the regulator can set any voltage within constrains range.

n_voltages Number of selectors available for ops.:c:func:*list_voltage()*.

ops Regulator operations table.

irq Interrupt number for the regulator.

type Indicates if the regulator is a voltage or current regulator.

owner Module providing the regulator, used for refcounting.

min_uV Voltage given by the lowest selector (if linear mapping)

uV_step Voltage increase with each selector (if linear mapping)

linear_min_sel Minimal selector for starting linear mapping

fixed_uV Fixed voltage of rails.

ramp_delay Time to settle down after voltage change (unit: uV/us)

min_dropout_uV The minimum dropout voltage this regulator can handle

linear_ranges A constant table of possible voltage ranges.

n_linear_ranges Number of entries in the **linear_ranges** table.

volt_table Voltage mapping table (if table based mapping)

vsel_reg Register for selector when using **regulator_regmap_X_voltage_**

vsel_mask Mask for register bitfield used for selector

csel_reg Register for TPS65218 LS3 current regulator

csel_mask Mask for TPS65218 LS3 current regulator

apply_reg Register for initiate voltage change on the output when using regulator_set_voltage_sel_regmap

apply_bit Register bitfield used for initiate voltage change on the output when using regulator_set_voltage_sel_regmap

enable_reg Register for control when using regmap enable/disable ops

enable_mask Mask for control when using regmap enable/disable ops

enable_val Enabling value for control when using regmap enable/disable ops

disable_val Disabling value for control when using regmap enable/disable ops

enable_is_inverted A flag to indicate set enable_mask bits to disable when using regulator_enable_regmap and friends APIs.

bypass_reg Register for control when using regmap set_bypass

bypass_mask Mask for control when using regmap set_bypass

bypass_val_on Enabling value for control when using regmap set_bypass

bypass_val_off Disabling value for control when using regmap set_bypass

active_discharge_on Disabling value for control when using regmap set_active_discharge

active_discharge_off Enabling value for control when using regmap set_active_discharge

active_discharge_mask Mask for control when using regmap set_active_discharge

active_discharge_reg Register for control when using regmap set_active_discharge

soft_start_reg Register for control when using regmap set_soft_start

soft_start_mask Mask for control when using regmap set_soft_start

soft_start_val_on Enabling value for control when using regmap set_soft_start

pull_down_reg Register for control when using regmap set_pull_down

pull_down_mask Mask for control when using regmap set_pull_down

pull_down_val_on Enabling value for control when using regmap set_pull_down

enable_time Time taken for initial enable of regulator (in uS).

off_on_delay guard time (in uS), before re-enabling a regulator

of_map_mode Maps a hardware mode defined in a DeviceTree to a standard mode

Description

Each regulator registered with the core is described with a structure of this type and a struct regulator_config. This structure contains the non-varying parts of the regulator description.

struct **regulator_config**
Dynamic regulator descriptor

Definition

```
struct regulator_config {
  struct device *dev;
  const struct regulator_init_data *init_data;
  void *driver_data;
  struct device_node *of_node;
  struct regmap *regmap;
  bool ena_gpio_initialized;
  int ena_gpio;
  unsigned int ena_gpio_invert:1;
  unsigned int ena_gpio_flags;
};
```

Members

dev struct device for the regulator

init_data platform provided init data, passed through by driver

driver_data private regulator data

of_node OpenFirmware node to parse for device tree bindings (may be NULL).

regmap regmap to use for core regmap helpers if `dev_get_regmap()` is insufficient.

ena_gpio_initialized GPIO controlling regulator enable was properly initialized, meaning that >= 0 is a valid gpio identifier and < 0 is a non existent gpio.

ena_gpio GPIO controlling regulator enable.

ena_gpio_invert Sense for GPIO enable control.

ena_gpio_flags Flags to use when calling *gpio_request_one()*

Description

Each regulator registered with the core is described with a structure of this type and a struct regulator_desc. This structure contains the runtime variable parts of the regulator description.

struct regulator * **regulator_get**(struct *device* * *dev*, const char * *id*)
lookup and obtain a reference to a regulator.

Parameters

struct device * dev device for regulator "consumer"

const char * id Supply name or regulator ID.

Description

Returns a struct regulator corresponding to the regulator producer, or `IS_ERR()` condition containing errno.

Use of supply names configured via `regulator_set_device_supply()` is strongly encouraged. It is recommended that the supply name used should match the name used for the supply and/or the relevant device pins in the datasheet.

struct regulator * **`regulator_get_exclusive`**(struct *device* * *dev*, const char * *id*)
obtain exclusive access to a regulator.

Parameters

`struct device * dev` device for regulator "consumer"

`const char * id` Supply name or regulator ID.

Description

Returns a struct regulator corresponding to the regulator producer, or `IS_ERR()` condition containing errno. Other consumers will be unable to obtain this regulator while this reference is held and the use count for the regulator will be initialised to reflect the current state of the regulator.

This is intended for use by consumers which cannot tolerate shared use of the regulator such as those which need to force the regulator off for correct operation of the hardware they are controlling.

Use of supply names configured via `regulator_set_device_supply()` is strongly encouraged. It is recommended that the supply name used should match the name used for the supply and/or the relevant device pins in the datasheet.

struct regulator * **`regulator_get_optional`**(struct *device* * *dev*, const char * *id*)
obtain optional access to a regulator.

Parameters

`struct device * dev` device for regulator "consumer"

`const char * id` Supply name or regulator ID.

Description

Returns a struct regulator corresponding to the regulator producer, or `IS_ERR()` condition containing errno.

This is intended for use by consumers for devices which can have some supplies unconnected in normal use, such as some MMC devices. It can allow the regulator core to provide stub supplies for other supplies requested using normal *`regulator_get()`* calls without disrupting the operation of drivers that can handle absent supplies.

Use of supply names configured via `regulator_set_device_supply()` is strongly encouraged. It is recommended that the supply name used should match the name used for the supply and/or the relevant device pins in the datasheet.

void **`regulator_put`**(struct regulator * *regulator*)
"free" the regulator source

Parameters

`struct regulator * regulator` regulator source

Note

drivers must ensure that all regulator_enable calls made on this regulator source are balanced by regulator_disable calls prior to calling this function.

int **`regulator_register_supply_alias`**(struct *device* * *dev*, const char * *id*, struct *device* * *alias_dev*, const char * *alias_id*)
Provide device alias for supply lookup

Parameters

`struct device * dev` device that will be given as the regulator "consumer"

`const char * id` Supply name or regulator ID

`struct device * alias_dev` device that should be used to lookup the supply

`const char * alias_id` Supply name or regulator ID that should be used to lookup the supply

Description

All lookups for id on dev will instead be conducted for alias_id on alias_dev.

void **regulator_unregister_supply_alias**(struct *device* * *dev*, const char * *id*)
Remove device alias

Parameters

struct device * dev device that will be given as the regulator "consumer"

const char * id Supply name or regulator ID

Description

Remove a lookup alias if one exists for id on dev.

int **regulator_bulk_register_supply_alias**(struct *device* * *dev*, const char *const * *id*, struct *device* * *alias_dev*, const char *const * *alias_id*, int *num_id*)
register multiple aliases

Parameters

struct device * dev device that will be given as the regulator "consumer"

const char *const * id List of supply names or regulator IDs

struct device * alias_dev device that should be used to lookup the supply

const char *const * alias_id List of supply names or regulator IDs that should be used to lookup the supply

int num_id Number of aliases to register

Description

return 0 on success, an errno on failure.

This helper function allows drivers to register several supply aliases in one operation. If any of the aliases cannot be registered any aliases that were registered will be removed before returning to the caller.

void **regulator_bulk_unregister_supply_alias**(struct *device* * *dev*, const char *const * *id*, int *num_id*)
unregister multiple aliases

Parameters

struct device * dev device that will be given as the regulator "consumer"

const char *const * id List of supply names or regulator IDs

int num_id Number of aliases to unregister

Description

This helper function allows drivers to unregister several supply aliases in one operation.

int **regulator_enable**(struct regulator * *regulator*)
enable regulator output

Parameters

struct regulator * regulator regulator source

Description

Request that the regulator be enabled with the regulator output at the predefined voltage or current value. Calls to `regulator_enable()` must be balanced with calls to `regulator_disable()`.

NOTE

the output value can be set by other drivers, boot loader or may be hardwired in the regulator.

int **regulator_disable**(struct regulator * *regulator*)
disable regulator output

Parameters

struct regulator * regulator regulator source

Description

Disable the regulator output voltage or current. Calls to *regulator_enable()* must be balanced with calls to *regulator_disable()*.

NOTE

this will only disable the regulator output if no other consumer devices have it enabled, the regulator device supports disabling and machine constraints permit this operation.

int **regulator_force_disable**(struct regulator * *regulator*)
force disable regulator output

Parameters

struct regulator * regulator regulator source

Description

Forcibly disable the regulator output voltage or current.

NOTE

this *will* disable the regulator output even if other consumer devices have it enabled. This should be used for situations when device damage will likely occur if the regulator is not disabled (e.g. over temp).

int **regulator_disable_deferred**(struct regulator * *regulator*, int *ms*)
disable regulator output with delay

Parameters

struct regulator * regulator regulator source

int ms miliseconds until the regulator is disabled

Description

Execute *regulator_disable()* on the regulator after a delay. This is intended for use with devices that require some time to quiesce.

NOTE

this will only disable the regulator output if no other consumer devices have it enabled, the regulator device supports disabling and machine constraints permit this operation.

int **regulator_is_enabled**(struct regulator * *regulator*)
is the regulator output enabled

Parameters

struct regulator * regulator regulator source

Description

Returns positive if the regulator driver backing the source/client has requested that the device be enabled, zero if it hasn't, else a negative errno code.

Note that the device backing this regulator handle can have multiple users, so it might be enabled even if *regulator_enable()* was never called for this particular source.

int **regulator_count_voltages**(struct regulator * *regulator*)
count *regulator_list_voltage()* selectors

Parameters

struct regulator * regulator regulator source

Description

Returns number of selectors, or negative errno. Selectors are numbered starting at zero, and typically correspond to bitfields in hardware registers.

int **`regulator_list_voltage`**(struct regulator * *regulator*, unsigned *selector*)
enumerate supported voltages

Parameters

`struct regulator * regulator` regulator source

`unsigned selector` identify voltage to list

Context

can sleep

Description

Returns a voltage that can be passed to **regulator_set_voltage()**, zero if this selector code can't be used on this system, or a negative errno.

int **`regulator_get_hardware_vsel_register`**(struct regulator * *regulator*, unsigned * *vsel_reg*, unsigned * *vsel_mask*)
get the HW voltage selector register

Parameters

`struct regulator * regulator` regulator source

`unsigned * vsel_reg` voltage selector register, output parameter

`unsigned * vsel_mask` mask for voltage selector bitfield, output parameter

Description

Returns the hardware register offset and bitmask used for setting the regulator voltage. This might be useful when configuring voltage-scaling hardware or firmware that can make I2C requests behind the kernel's back, for example.

On success, the output parameters **vsel_reg** and **vsel_mask** are filled in and 0 is returned, otherwise a negative errno is returned.

int **`regulator_list_hardware_vsel`**(struct regulator * *regulator*, unsigned *selector*)
get the HW-specific register value for a selector

Parameters

`struct regulator * regulator` regulator source

`unsigned selector` identify voltage to list

Description

Converts the selector to a hardware-specific voltage selector that can be directly written to the regulator registers. The address of the voltage register can be determined by calling **regulator_get_hardware_vsel_register**.

On error a negative errno is returned.

unsigned int **`regulator_get_linear_step`**(struct regulator * *regulator*)
return the voltage step size between VSEL values

Parameters

`struct regulator * regulator` regulator source

Description

Returns the voltage step size between VSEL values for linear regulators, or return 0 if the regulator isn't a linear regulator.

int **regulator_is_supported_voltage**(struct regulator * *regulator*, int *min_uV*, int *max_uV*)
check if a voltage range can be supported

Parameters

struct regulator * regulator Regulator to check.

int min_uV Minimum required voltage in uV.

int max_uV Maximum required voltage in uV.

Description

Returns a boolean or a negative error code.

int **regulator_set_voltage**(struct regulator * *regulator*, int *min_uV*, int *max_uV*)
set regulator output voltage

Parameters

struct regulator * regulator regulator source

int min_uV Minimum required voltage in uV

int max_uV Maximum acceptable voltage in uV

Description

Sets a voltage regulator to the desired output voltage. This can be set during any regulator state. IOW, regulator can be disabled or enabled.

If the regulator is enabled then the voltage will change to the new value immediately otherwise if the regulator is disabled the regulator will output at the new voltage when enabled.

NOTE

If the regulator is shared between several devices then the lowest request voltage that meets the system constraints will be used. Regulator system constraints must be set for this regulator before calling this function otherwise this call will fail.

int **regulator_set_voltage_time**(struct regulator * *regulator*, int *old_uV*, int *new_uV*)
get raise/fall time

Parameters

struct regulator * regulator regulator source

int old_uV starting voltage in microvolts

int new_uV target voltage in microvolts

Description

Provided with the starting and ending voltage, this function attempts to calculate the time in microseconds required to rise or fall to this new voltage.

int **regulator_set_voltage_time_sel**(struct regulator_dev * *rdev*, unsigned int *old_selector*, unsigned int *new_selector*)
get raise/fall time

Parameters

struct regulator_dev * rdev regulator source device

unsigned int old_selector selector for starting voltage

unsigned int new_selector selector for target voltage

Description

Provided with the starting and target voltage selectors, this function returns time in microseconds required to rise or fall to this new voltage

Drivers providing ramp_delay in regulation_constraints can use this as their `set_voltage_time_sel()` operation.

int **`regulator_sync_voltage`**(struct regulator * *regulator*)
re-apply last regulator output voltage

Parameters

`struct regulator * regulator` regulator source

Description

Re-apply the last configured voltage. This is intended to be used where some external control source the consumer is cooperating with has caused the configured voltage to change.

int **`regulator_get_voltage`**(struct regulator * *regulator*)
get regulator output voltage

Parameters

`struct regulator * regulator` regulator source

Description

This returns the current regulator voltage in uV.

NOTE

If the regulator is disabled it will return the voltage value. This function should not be used to determine regulator state.

int **`regulator_set_current_limit`**(struct regulator * *regulator*, int *min_uA*, int *max_uA*)
set regulator output current limit

Parameters

`struct regulator * regulator` regulator source

`int min_uA` Minimum supported current in uA

`int max_uA` Maximum supported current in uA

Description

Sets current sink to the desired output current. This can be set during any regulator state. IOW, regulator can be disabled or enabled.

If the regulator is enabled then the current will change to the new value immediately otherwise if the regulator is disabled the regulator will output at the new current when enabled.

NOTE

Regulator system constraints must be set for this regulator before calling this function otherwise this call will fail.

int **`regulator_get_current_limit`**(struct regulator * *regulator*)
get regulator output current

Parameters

`struct regulator * regulator` regulator source

Description

This returns the current supplied by the specified current sink in uA.

NOTE

If the regulator is disabled it will return the current value. This function should not be used to determine regulator state.

int **`regulator_set_mode`**(struct regulator * *regulator*, unsigned int *mode*)
set regulator operating mode

Parameters

`struct regulator * regulator` regulator source

`unsigned int mode` operating mode - one of the REGULATOR_MODE constants

Description

Set regulator operating mode to increase regulator efficiency or improve regulation performance.

NOTE

Regulator system constraints must be set for this regulator before calling this function otherwise this call will fail.

unsigned int **`regulator_get_mode`**(struct regulator * *regulator*)
get regulator operating mode

Parameters

`struct regulator * regulator` regulator source

Description

Get the current regulator operating mode.

int **`regulator_get_error_flags`**(struct regulator * *regulator*, unsigned int * *flags*)
get regulator error information

Parameters

`struct regulator * regulator` regulator source

`unsigned int * flags` pointer to store error flags

Description

Get the current regulator error information.

int **`regulator_set_load`**(struct regulator * *regulator*, int *uA_load*)
set regulator load

Parameters

`struct regulator * regulator` regulator source

`int uA_load` load current

Description

Notifies the regulator core of a new device load. This is then used by DRMS (if enabled by constraints) to set the most efficient regulator operating mode for the new regulator loading.

Consumer devices notify their supply regulator of the maximum power they will require (can be taken from device datasheet in the power consumption tables) when they change operational status and hence power state. Examples of operational state changes that can affect power consumption are :-

> o Device is opened / closed. o Device I/O is about to begin or has just finished. o Device is idling in between work.

This information is also exported via sysfs to userspace.

DRMS will sum the total requested load on the regulator and change to the most efficient operating mode if platform constraints allow.

On error a negative errno is returned.

int **`regulator_allow_bypass`**(struct regulator * *regulator*, bool *enable*)
allow the regulator to go into bypass mode

Parameters

`struct regulator * regulator` Regulator to configure

bool enable enable or disable bypass mode

Description

Allow the regulator to go into bypass mode if all other consumers for the regulator also enable bypass mode and the machine constraints allow this. Bypass mode means that the regulator is simply passing the input directly to the output with no regulation.

int **regulator_register_notifier**(struct regulator * *regulator*, struct notifier_block * *nb*)
register regulator event notifier

Parameters

struct regulator * regulator regulator source

struct notifier_block * nb notifier block

Description

Register notifier block to receive regulator events.

int **regulator_unregister_notifier**(struct regulator * *regulator*, struct notifier_block * *nb*)
unregister regulator event notifier

Parameters

struct regulator * regulator regulator source

struct notifier_block * nb notifier block

Description

Unregister regulator event notifier block.

int **regulator_bulk_get**(struct *device* * *dev*, int *num_consumers*, struct *regulator_bulk_data* * *consumers*)
get multiple regulator consumers

Parameters

struct device * dev Device to supply

int num_consumers Number of consumers to register

struct regulator_bulk_data * consumers Configuration of consumers; clients are stored here.

Description

return 0 on success, an errno on failure.

This helper function allows drivers to get several regulator consumers in one operation. If any of the regulators cannot be acquired then any regulators that were allocated will be freed before returning to the caller.

int **regulator_bulk_enable**(int *num_consumers*, struct *regulator_bulk_data* * *consumers*)
enable multiple regulator consumers

Parameters

int num_consumers Number of consumers

struct regulator_bulk_data * consumers Consumer data; clients are stored here. **return** 0 on success, an errno on failure

Description

This convenience API allows consumers to enable multiple regulator clients in a single API call. If any consumers cannot be enabled then any others that were enabled will be disabled again prior to return.

int **regulator_bulk_disable**(int *num_consumers*, struct *regulator_bulk_data* * *consumers*)
disable multiple regulator consumers

Parameters

int num_consumers Number of consumers

struct regulator_bulk_data * consumers Consumer data; clients are stored here. **return** 0 on success, an errno on failure

Description

This convenience API allows consumers to disable multiple regulator clients in a single API call. If any consumers cannot be disabled then any others that were disabled will be enabled again prior to return.

int **regulator_bulk_force_disable**(int *num_consumers*, struct *regulator_bulk_data* * *consumers*)
force disable multiple regulator consumers

Parameters

int num_consumers Number of consumers

struct regulator_bulk_data * consumers Consumer data; clients are stored here. **return** 0 on success, an errno on failure

Description

This convenience API allows consumers to forcibly disable multiple regulator clients in a single API call.

NOTE

This should be used for situations when device damage will likely occur if the regulators are not disabled (e.g. over temp). Although regulator_force_disable function call for some consumers can return error numbers, the function is called for all consumers.

void **regulator_bulk_free**(int *num_consumers*, struct *regulator_bulk_data* * *consumers*)
free multiple regulator consumers

Parameters

int num_consumers Number of consumers

struct regulator_bulk_data * consumers Consumer data; clients are stored here.

Description

This convenience API allows consumers to free multiple regulator clients in a single API call.

int **regulator_notifier_call_chain**(struct regulator_dev * *rdev*, unsigned long *event*, void * *data*)
call regulator event notifier

Parameters

struct regulator_dev * rdev regulator source

unsigned long event notifier block

void * data callback-specific data.

Description

Called by regulator drivers to notify clients a regulator event has occurred. We also notify regulator clients downstream. Note lock must be held by caller.

int **regulator_mode_to_status**(unsigned int *mode*)
convert a regulator mode into a status

Parameters

unsigned int mode Mode to convert

Description

Convert a regulator mode into a status.

struct regulator_dev * **regulator_register**(const struct *regulator_desc* * *regulator_desc*, const struct *regulator_config* * *cfg*)
register regulator

Parameters

const struct regulator_desc * regulator_desc regulator to register

const struct regulator_config * cfg runtime configuration for regulator

Description

Called by regulator drivers to register a regulator. Returns a valid pointer to struct regulator_dev on success or an ERR_PTR() on error.

void **regulator_unregister**(struct regulator_dev * *rdev*)
unregister regulator

Parameters

struct regulator_dev * rdev regulator to unregister

Description

Called by regulator drivers to unregister a regulator.

void **regulator_has_full_constraints**(void)
the system has fully specified constraints

Parameters

void no arguments

Description

Calling this function will cause the regulator API to disable all regulators which have a zero use count and don't have an always_on constraint in a late_initcall.

The intention is that this will become the default behaviour in a future kernel release so users are encouraged to use this facility now.

void * **rdev_get_drvdata**(struct regulator_dev * *rdev*)
get rdev regulator driver data

Parameters

struct regulator_dev * rdev regulator

Description

Get rdev regulator driver private data. This call can be used in the regulator driver context.

void * **regulator_get_drvdata**(struct regulator * *regulator*)
get regulator driver data

Parameters

struct regulator * regulator regulator

Description

Get regulator driver private data. This call can be used in the consumer driver context when non API regulator specific functions need to be called.

void **regulator_set_drvdata**(struct regulator * *regulator*, void * *data*)
set regulator driver data

Parameters

struct regulator * regulator regulator

void * data data

int **rdev_get_id**(struct regulator_dev * *rdev*)
get regulator ID

Parameters

struct regulator_dev * rdev regulator

CHAPTER

ELEVEN

INDUSTRIAL I/O

Contents:

Introduction

The main purpose of the Industrial I/O subsystem (IIO) is to provide support for devices that in some sense perform either analog-to-digital conversion (ADC) or digital-to-analog conversion (DAC) or both. The aim is to fill the gap between the somewhat similar hwmon and input subsystems. Hwmon is directed at low sample rate sensors used to monitor and control the system itself, like fan speed control or temperature measurement. Input is, as its name suggests, focused on human interaction input devices (keyboard, mouse, touchscreen). In some cases there is considerable overlap between these and IIO.

Devices that fall into this category include:

- analog to digital converters (ADCs)
- accelerometers
- capacitance to digital converters (CDCs)
- digital to analog converters (DACs)
- gyroscopes
- inertial measurement units (IMUs)
- color and light sensors
- magnetometers
- pressure sensors
- proximity sensors
- temperature sensors

Usually these sensors are connected via SPI or I2C. A common use case of the sensors devices is to have combined functionality (e.g. light plus proximity sensor).

Core elements

The Industrial I/O core offers a unified framework for writing drivers for many different types of embedded sensors. a standard interface to user space applications manipulating sensors. The implementation can be found under `drivers/iio/industrialio-*`

Industrial I/O Devices

- struct *iio_dev* - industrial I/O device
- *iio_device_alloc()* - alocate an *iio_dev* from a driver
- *iio_device_free()* - free an *iio_dev* from a driver
- *iio_device_register()* - register a device with the IIO subsystem
- *iio_device_unregister()* - unregister a device from the IIO subsystem

An IIO device usually corresponds to a single hardware sensor and it provides all the information needed by a driver handling a device. Let's first have a look at the functionality embedded in an IIO device then we will show how a device driver makes use of an IIO device.

There are two ways for a user space application to interact with an IIO driver.

1. `/sys/bus/iio/iio:device`*X*`/`, this represents a hardware sensor and groups together the data channels of the same chip.
2. `/dev/iio:device`*X*, character device node interface used for buffered data transfer and for events information retrieval.

A typical IIO driver will register itself as an I2C or SPI driver and will create two routines, probe and remove.

At probe:

1. Call *iio_device_alloc()*, which allocates memory for an IIO device.
2. Initialize IIO device fields with driver specific information (e.g. device name, device channels).
3. Call *iio_device_register()*, this registers the device with the IIO core. After this call the device is ready to accept requests from user space applications.

At remove, we free the resources allocated in probe in reverse order:

1. *iio_device_unregister()*, unregister the device from the IIO core.
2. *iio_device_free()*, free the memory allocated for the IIO device.

IIO device sysfs interface

Attributes are sysfs files used to expose chip info and also allowing applications to set various configuration parameters. For device with index X, attributes can be found under /sys/bus/iio/iio:deviceX/ directory. Common attributes are:

- name, description of the physical chip.
- dev, shows the major:minor pair associated with `/dev/iio:deviceX` node.
- `sampling_frequency_available`, available discrete set of sampling frequency values for device.
- Available standard attributes for IIO devices are described in the `Documentation/ABI/testing/sysfs-bus-iio` file in the Linux kernel sources.

IIO device channels

struct *iio_chan_spec* - specification of a single channel

An IIO device channel is a representation of a data channel. An IIO device can have one or multiple channels. For example:

- a thermometer sensor has one channel representing the temperature measurement.
- a light sensor with two channels indicating the measurements in the visible and infrared spectrum.
- an accelerometer can have up to 3 channels representing acceleration on X, Y and Z axes.

An IIO channel is described by the struct *iio_chan_spec*. A thermometer driver for the temperature sensor in the example above would have to describe its channel as follows:

```
static const struct iio_chan_spec temp_channel[] = {
     {
         .type = IIO_TEMP,
         .info_mask_separate = BIT(IIO_CHAN_INFO_PROCESSED),
     },
};
```

Channel sysfs attributes exposed to userspace are specified in the form of bitmasks. Depending on their shared info, attributes can be set in one of the following masks:

- **info_mask_separate**, attributes will be specific to this channel
- **info_mask_shared_by_type**, attributes are shared by all channels of the same type
- **info_mask_shared_by_dir**, attributes are shared by all channels of the same direction
- **info_mask_shared_by_all**, attributes are shared by all channels

When there are multiple data channels per channel type we have two ways to distinguish between them:

- set **.modified** field of *iio_chan_spec* to 1. Modifiers are specified using **.channel2** field of the same *iio_chan_spec* structure and are used to indicate a physically unique characteristic of the channel such as its direction or spectral response. For example, a light sensor can have two channels, one for infrared light and one for both infrared and visible light.
- set **.indexed** field of *iio_chan_spec* to 1. In this case the channel is simply another instance with an index specified by the **.channel** field.

Here is how we can make use of the channel's modifiers:

```
static const struct iio_chan_spec light_channels[] = {
        {
                .type = IIO_INTENSITY,
                .modified = 1,
                .channel2 = IIO_MOD_LIGHT_IR,
                .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
                .info_mask_shared = BIT(IIO_CHAN_INFO_SAMP_FREQ),
        },
        {
                .type = IIO_INTENSITY,
                .modified = 1,
                .channel2 = IIO_MOD_LIGHT_BOTH,
                .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
                .info_mask_shared = BIT(IIO_CHAN_INFO_SAMP_FREQ),
        },
        {
                .type = IIO_LIGHT,
                .info_mask_separate = BIT(IIO_CHAN_INFO_PROCESSED),
                .info_mask_shared = BIT(IIO_CHAN_INFO_SAMP_FREQ),
        },
   }
```

This channel's definition will generate two separate sysfs files for raw data retrieval:

- `/sys/bus/iio/iio:device`*X*`/in_intensity_ir_raw`
- `/sys/bus/iio/iio:device`*X*`/in_intensity_both_raw`

one file for processed data:

- `/sys/bus/iio/iio:device`*X*`/in_illuminance_input`

and one shared sysfs file for sampling frequency:

- `/sys/bus/iio/iio:device`*X*`/sampling_frequency`.

Here is how we can make use of the channel's indexing:

```
static const struct iio_chan_spec light_channels[] = {
        {
                .type = IIO_VOLTAGE,
                .indexed = 1,
                .channel = 0,
                .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
        },
        {
                .type = IIO_VOLTAGE,
                .indexed = 1,
                .channel = 1,
                .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
        },
}
```

This will generate two separate attributes files for raw data retrieval:

- `/sys/bus/iio/devices/iio:device`*X*`/in_voltage0_raw`, representing voltage measurement for channel 0.
- `/sys/bus/iio/devices/iio:device`*X*`/in_voltage1_raw`, representing voltage measurement for channel 1.

More details

struct **`iio_chan_spec_ext_info`**
Extended channel info attribute

Definition

```
struct iio_chan_spec_ext_info {
  const char *name;
  enum iio_shared_by shared;
  ssize_t (*read)(struct iio_dev *, uintptr_t private, struct iio_chan_spec const *, char *buf);
  ssize_t (*write)(struct iio_dev *, uintptr_t private,struct iio_chan_spec const *, const char *buf, si
  uintptr_t private;
};
```

Members

`name` Info attribute name

`shared` Whether this attribute is shared between all channels.

`read` Read callback for this info attribute, may be NULL.

`write` Write callback for this info attribute, may be NULL.

`private` Data private to the driver.

struct **`iio_enum`**
Enum channel info attribute

Definition

```
struct iio_enum {
  const char * const *items;
  unsigned int num_items;
  int (*set)(struct iio_dev *, const struct iio_chan_spec *, unsigned int);
  int (*get)(struct iio_dev *, const struct iio_chan_spec *);
};
```

Members

items An array of strings.

num_items Length of the item array.

set Set callback function, may be NULL.

get Get callback function, may be NULL.

Description

The iio_enum struct can be used to implement enum style channel attributes. Enum style attributes are those which have a set of strings which map to unsigned integer values. The IIO enum helper code takes care of mapping between value and string as well as generating a "_available" file which contains a list of all available items. The set callback will be called when the attribute is updated. The last parameter is the index to the newly activated item. The get callback will be used to query the currently active item and is supposed to return the index for it.

IIO_ENUM(*_name*, *_shared*, *_e*)
Initialize enum extended channel attribute

Parameters

_name Attribute name

_shared Whether the attribute is shared between all channels

_e Pointer to an iio_enum struct

Description

This should usually be used together with *IIO_ENUM_AVAILABLE()*

IIO_ENUM_AVAILABLE(*_name*, *_e*)
Initialize enum available extended channel attribute

Parameters

_name Attribute name ("_available" will be appended to the name)

_e Pointer to an iio_enum struct

Description

Creates a read only attribute which lists all the available enum items in a space separated list. This should usually be used together with *IIO_ENUM()*

struct **iio_mount_matrix**
iio mounting matrix

Definition

```
struct iio_mount_matrix {
  const char *rotation[9];
};
```

Members

rotation 3 dimensional space rotation matrix defining sensor alignment with main hardware

IIO_MOUNT_MATRIX(*_shared*, *_get*)
Initialize mount matrix extended channel attribute

Parameters

_shared Whether the attribute is shared between all channels

_get Pointer to an iio_get_mount_matrix_t accessor

struct **iio_event_spec**
specification for a channel event

Definition

```
struct iio_event_spec {
  enum iio_event_type type;
  enum iio_event_direction dir;
  unsigned long mask_separate;
  unsigned long mask_shared_by_type;
  unsigned long mask_shared_by_dir;
  unsigned long mask_shared_by_all;
};
```

Members

type Type of the event

dir Direction of the event

mask_separate Bit mask of enum iio_event_info values. Attributes set in this mask will be registered per channel.

mask_shared_by_type Bit mask of enum iio_event_info values. Attributes set in this mask will be shared by channel type.

mask_shared_by_dir Bit mask of enum iio_event_info values. Attributes set in this mask will be shared by channel type and direction.

mask_shared_by_all Bit mask of enum iio_event_info values. Attributes set in this mask will be shared by all channels.

struct **iio_chan_spec**
specification of a single channel

Definition

```
struct iio_chan_spec {
  enum iio_chan_type      type;
  int channel;
  int channel2;
  unsigned long           address;
  int scan_index;
  struct {
    char sign;
    u8 realbits;
    u8 storagebits;
    u8 shift;
    u8 repeat;
    enum iio_endian endianness;
  } scan_type;
  long info_mask_separate;
  long info_mask_separate_available;
  long info_mask_shared_by_type;
  long info_mask_shared_by_type_available;
  long info_mask_shared_by_dir;
  long info_mask_shared_by_dir_available;
  long info_mask_shared_by_all;
  long info_mask_shared_by_all_available;
  const struct iio_event_spec *event_spec;
  unsigned int            num_event_specs;
  const struct iio_chan_spec_ext_info *ext_info;
  const char              *extend_name;
  const char              *datasheet_name;
  unsigned modified:1;
  unsigned indexed:1;
  unsigned output:1;
  unsigned differential:1;
};
```

Members

type What type of measurement is the channel making.

channel What number do we wish to assign the channel.

channel2 If there is a second number for a differential channel then this is it. If modified is set then the value here specifies the modifier.

address Driver specific identifier.

scan_index Monotonic index to give ordering in scans when read from a buffer.

scan_type sign: 's' or 'u' to specify signed or unsigned realbits: Number of valid bits of data storagebits: Realbits + padding shift: Shift right by this before masking out

realbits.

repeat: Number of times real/storage bits repeats. When the repeat element is more than 1, then the type element in sysfs will show a repeat value. Otherwise, the number of repetitions is omitted.

endianness: little or big endian

info_mask_separate What information is to be exported that is specific to this channel.

info_mask_separate_available What availability information is to be exported that is specific to this channel.

info_mask_shared_by_type What information is to be exported that is shared by all channels of the same type.

info_mask_shared_by_type_available What availability information is to be exported that is shared by all channels of the same type.

info_mask_shared_by_dir What information is to be exported that is shared by all channels of the same direction.

info_mask_shared_by_dir_available What availability information is to be exported that is shared by all channels of the same direction.

info_mask_shared_by_all What information is to be exported that is shared by all channels.

info_mask_shared_by_all_available What availability information is to be exported that is shared by all channels.

event_spec Array of events which should be registered for this channel.

num_event_specs Size of the event_spec array.

ext_info Array of extended info attributes for this channel. The array is NULL terminated, the last element should have its name field set to NULL.

extend_name Allows labeling of channel attributes with an informative name. Note this has no effect codes etc, unlike modifiers.

datasheet_name A name used in in-kernel mapping of channels. It should correspond to the first name that the channel is referred to by in the datasheet (e.g. IND), or the nearest possible compound name (e.g. IND-INC).

modified Does a modifier apply to this channel. What these are depends on the channel type. Modifier is set in channel2. Examples are IIO_MOD_X for axial sensors about the 'x' axis.

indexed Specify the channel has a numerical index. If not, the channel index number will be suppressed for sysfs attributes but not for event codes.

output Channel is output.

differential Channel is differential.

bool **iio_channel_has_info**(const struct *iio_chan_spec* * *chan*, enum iio_chan_info_enum *type*)
Checks whether a channel supports a info attribute

Parameters

const struct iio_chan_spec * chan The channel to be queried

enum iio_chan_info_enum type Type of the info attribute to be checked

Description

Returns true if the channels supports reporting values for the given info attribute type, false otherwise.

bool **iio_channel_has_available**(const struct *iio_chan_spec* * *chan*, enum iio_chan_info_enum *type*)
Checks if a channel has an available attribute

Parameters

const struct iio_chan_spec * chan The channel to be queried

enum iio_chan_info_enum type Type of the available attribute to be checked

Description

Returns true if the channel supports reporting available values for the given attribute type, false otherwise.

struct **iio_info**
constant information about device

Definition

```
struct iio_info {
  const struct attribute_group    *event_attrs;
  const struct attribute_group    *attrs;
  int (*read_raw)(struct iio_dev *indio_dev,struct iio_chan_spec const *chan,int *val,int *val2, long ma
  int (*read_raw_multi)(struct iio_dev *indio_dev,struct iio_chan_spec const *chan,int max_len,int *vals
  int (*read_avail)(struct iio_dev *indio_dev,struct iio_chan_spec const *chan,const int **vals,int *typ
  int (*write_raw)(struct iio_dev *indio_dev,struct iio_chan_spec const *chan,int val,int val2, long mas
  int (*write_raw_get_fmt)(struct iio_dev *indio_dev,struct iio_chan_spec const *chan, long mask);
  int (*read_event_config)(struct iio_dev *indio_dev,const struct iio_chan_spec *chan,enum iio_event_typ
  int (*write_event_config)(struct iio_dev *indio_dev,const struct iio_chan_spec *chan,enum iio_event_ty
  int (*read_event_value)(struct iio_dev *indio_dev,const struct iio_chan_spec *chan,enum iio_event_type
  int (*write_event_value)(struct iio_dev *indio_dev,const struct iio_chan_spec *chan,enum iio_event_typ
  int (*validate_trigger)(struct iio_dev *indio_dev, struct iio_trigger *trig);
  int (*update_scan_mode)(struct iio_dev *indio_dev, const unsigned long *scan_mask);
  int (*debugfs_reg_access)(struct iio_dev *indio_dev,unsigned reg, unsigned writeval, unsigned *readval
  int (*of_xlate)(struct iio_dev *indio_dev, const struct of_phandle_args *iiospec);
  int (*hwfifo_set_watermark)(struct iio_dev *indio_dev, unsigned val);
  int (*hwfifo_flush_to_buffer)(struct iio_dev *indio_dev, unsigned count);
};
```

Members

event_attrs event control attributes

attrs general purpose device attributes

read_raw function to request a value from the device. mask specifies which value. Note 0 means a reading of the channel in question. Return value will specify the type of value returned by the device. val and val2 will contain the elements making up the returned value.

read_raw_multi function to return values from the device. mask specifies which value. Note 0 means a reading of the channel in question. Return value will specify the type of value returned by the device. vals pointer contain the elements making up the returned value. max_len specifies maximum number of elements vals pointer can contain. val_len is used to return length of valid elements in vals.

read_avail function to return the available values from the device. mask specifies which value. Note 0 means the available values for the channel in question. Return value specifies if a IIO_AVAIL_LIST or a IIO_AVAIL_RANGE is returned in vals. The type of the vals are returned in type and the number of vals is returned in length. For ranges, there are always three vals returned; min, step and max. For lists, all possible values are enumerated.

write_raw function to write a value to the device. Parameters are the same as for read_raw.

write_raw_get_fmt callback function to query the expected format/precision. If not set by the driver, write_raw returns IIO_VAL_INT_PLUS_MICRO.

read_event_config find out if the event is enabled.

write_event_config set if the event is enabled.

read_event_value read a configuration value associated with the event.

write_event_value write a configuration value for the event.

validate_trigger function to validate the trigger when the current trigger gets changed.

update_scan_mode function to configure device and scan buffer when channels have changed

debugfs_reg_access function to read or write register value of device

of_xlate function pointer to obtain channel specifier index. When #iio-cells is greater than '0', the driver could provide a custom of_xlate function that reads the *args* and returns the appropriate index in registered IIO channels array.

hwfifo_set_watermark function pointer to set the current hardware fifo watermark level; see hwfifo_* entries in Documentation/ABI/testing/sysfs-bus-iio for details on how the hardware fifo operates

hwfifo_flush_to_buffer function pointer to flush the samples stored in the hardware fifo to the device buffer. The driver should not flush more than count samples. The function must return the number of samples flushed, 0 if no samples were flushed or a negative integer if no samples were flushed and there was an error.

struct **iio_buffer_setup_ops**
buffer setup related callbacks

Definition

```
struct iio_buffer_setup_ops {
  int (*preenable)(struct iio_dev *);
  int (*postenable)(struct iio_dev *);
  int (*predisable)(struct iio_dev *);
  int (*postdisable)(struct iio_dev *);
  bool (*validate_scan_mask)(struct iio_dev *indio_dev, const unsigned long *scan_mask);
};
```

Members

preenable [DRIVER] function to run prior to marking buffer enabled

postenable [DRIVER] function to run after marking buffer enabled

predisable [DRIVER] function to run prior to marking buffer disabled

postdisable [DRIVER] function to run after marking buffer disabled

validate_scan_mask [DRIVER] function callback to check whether a given scan mask is valid for the device.

struct **iio_dev**
industrial I/O device

Definition

```
struct iio_dev {
  int id;
  struct module                 *driver_module;
  int modes;
  int currentmode;
  struct device                 dev;
  struct iio_event_interface    *event_interface;
  struct iio_buffer             *buffer;
  struct list_head                buffer_list;
  int scan_bytes;
  struct mutex                    mlock;
  const unsigned long             *available_scan_masks;
  unsigned masklength;
  const unsigned long             *active_scan_mask;
  bool scan_timestamp;
  unsigned scan_index_timestamp;
  struct iio_trigger              *trig;
  bool trig_readonly;
  struct iio_poll_func            *pollfunc;
  struct iio_poll_func            *pollfunc_event;
  struct iio_chan_spec const      *channels;
  int num_channels;
  struct list_head                channel_attr_list;
  struct attribute_group          chan_attr_group;
  const char                      *name;
  const struct iio_info           *info;
  clockid_t clock_id;
  struct mutex                    info_exist_lock;
  const struct iio_buffer_setup_ops       *setup_ops;
  struct cdev                     chrdev;
#define IIO_MAX_GROUPS 6;
  const struct attribute_group    *groups[IIO_MAX_GROUPS + 1];
  int groupcounter;
  unsigned long                   flags;
#if defined(CONFIG_DEBUG_FS);
  struct dentry                   *debugfs_dentry;
  unsigned cached_reg_addr;
#endif;
};
```

Members

id [INTERN] used to identify device internally

driver_module [INTERN] used to make it harder to undercut users

modes [DRIVER] operating modes supported by device

currentmode [DRIVER] current operating mode

dev [DRIVER] device structure, should be assigned a parent and owner

event_interface [INTERN] event chrdevs associated with interrupt lines

buffer [DRIVER] any buffer present

buffer_list [INTERN] list of all buffers currently attached

scan_bytes [INTERN] num bytes captured to be fed to buffer demux

mlock [DRIVER] lock used to prevent simultaneous device state changes

available_scan_masks [DRIVER] optional array of allowed bitmasks

masklength [INTERN] the length of the mask established from channels

active_scan_mask [INTERN] union of all scan masks requested by buffers

scan_timestamp [INTERN] set if any buffers have requested timestamp

scan_index_timestamp [INTERN] cache of the index to the timestamp

trig [INTERN] current device trigger (buffer modes)

trig_readonly [INTERN] mark the current trigger immutable

pollfunc [DRIVER] function run on trigger being received

pollfunc_event [DRIVER] function run on events trigger being received

channels [DRIVER] channel specification structure table

num_channels [DRIVER] number of channels specified in **channels**.

channel_attr_list [INTERN] keep track of automatically created channel attributes

chan_attr_group [INTERN] group for all attrs in base directory

name [DRIVER] name of the device.

info [DRIVER] callbacks and constant info from driver

clock_id [INTERN] timestamping clock posix identifier

info_exist_lock [INTERN] lock to prevent use during removal

setup_ops [DRIVER] callbacks to call before and after buffer enable/disable

chrdev [INTERN] associated character device

groups [INTERN] attribute groups

groupcounter [INTERN] index of next attribute group

flags [INTERN] file ops related flags including busy flag.

debugfs_dentry [INTERN] device specific debugfs dentry.

cached_reg_addr [INTERN] cached register address for debugfs reads.

iio_device_register(*indio_dev*)
register a device with the IIO subsystem

Parameters

indio_dev Device structure filled by the device driver

devm_iio_device_register(*dev*, *indio_dev*)
Resource-managed *iio_device_register()*

Parameters

dev Device to allocate iio_dev for

indio_dev Device structure filled by the device driver

Description

Managed iio_device_register. The IIO device registered with this function is automatically unregistered on driver detach. This function calls *iio_device_register()* internally. Refer to that function for more information.

If an iio_dev registered with this function needs to be unregistered separately, *devm_iio_device_unregister()* must be used.

Return

0 on success, negative error number on failure.

void **iio_device_put**(struct *iio_dev* * *indio_dev*)
reference counted deallocation of struct device

Parameters

struct iio_dev * indio_dev IIO device structure containing the device

clockid_t **iio_device_get_clock**(const struct *iio_dev* * *indio_dev*)
Retrieve current timestamping clock for the device

Parameters

const struct iio_dev * indio_dev IIO device structure containing the device

struct *iio_dev* * **dev_to_iio_dev**(struct *device* * *dev*)
Get IIO device struct from a device struct

Parameters

struct device * dev The device embedded in the IIO device

Note

The device must be a IIO device, otherwise the result is undefined.

struct *iio_dev* * **iio_device_get**(struct *iio_dev* * *indio_dev*)
increment reference count for the device

Parameters

struct iio_dev * indio_dev IIO device structure

Return

The passed IIO device

void **iio_device_set_drvdata**(struct *iio_dev* * *indio_dev*, void * *data*)
Set device driver data

Parameters

struct iio_dev * indio_dev IIO device structure

void * data Driver specific data

Description

Allows to attach an arbitrary pointer to an IIO device, which can later be retrieved by *iio_device_get_drvdata()*.

void * **iio_device_get_drvdata**(struct *iio_dev* * *indio_dev*)
Get device driver data

Parameters

struct iio_dev * indio_dev IIO device structure

Description

Returns the data previously set with *iio_device_set_drvdata()*

bool **iio_buffer_enabled**(struct *iio_dev* * *indio_dev*)
helper function to test if the buffer is enabled

Parameters

struct iio_dev * indio_dev IIO device structure for device

struct dentry * **iio_get_debugfs_dentry**(struct *iio_dev* * *indio_dev*)
helper function to get the debugfs_dentry

Parameters

struct iio_dev * indio_dev IIO device structure for device

IIO_DEGREE_TO_RAD(*deg*)
Convert degree to rad

Parameters

deg A value in degree

Description

Returns the given value converted from degree to rad

IIO_RAD_TO_DEGREE(*rad*)
Convert rad to degree

Parameters

rad A value in rad

Description

Returns the given value converted from rad to degree

IIO_G_TO_M_S_2(*g*)
Convert g to meter / second**2

Parameters

g A value in g

Description

Returns the given value converted from g to meter / second**2

IIO_M_S_2_TO_G(*ms2*)
Convert meter / second**2 to g

Parameters

ms2 A value in meter / second**2

Description

Returns the given value converted from meter / second**2 to g

s64 **iio_get_time_ns**(const struct *iio_dev* * *indio_dev*)
utility function to get a time stamp for events etc

Parameters

const struct iio_dev * indio_dev device

unsigned int **iio_get_time_res**(const struct *iio_dev* * *indio_dev*)
utility function to get time stamp clock resolution in nano seconds.

Parameters

const struct iio_dev * indio_dev device

int **of_iio_read_mount_matrix**(const struct *device* * *dev*, const char * *propname*, struct *iio_mount_matrix* * *matrix*)
retrieve iio device mounting matrix from device-tree "mount-matrix" property

Parameters

const struct device * dev device the mounting matrix property is assigned to

const char * propname device specific mounting matrix property name

struct iio_mount_matrix * matrix where to store retrieved matrix

Description

If device is assigned no mounting matrix property, a default 3x3 identity matrix will be filled in.

Return

0 if success, or a negative error code on failure.

ssize_t **iio_format_value**(char * *buf*, unsigned int *type*, int *size*, int * *vals*)
Formats a IIO value into its string representation

Parameters

char * buf The buffer to which the formatted value gets written which is assumed to be big enough (i.e. PAGE_SIZE).

unsigned int type One of the IIO_VAL_* constants. This decides how the val and val2 parameters are formatted.

int size Number of IIO value entries contained in vals

int * vals Pointer to the values, exact meaning depends on the type parameter.

Return

0 by default, a negative number on failure or the total number of characters written for a type that belongs to the IIO_VAL_* constant.

int **iio_str_to_fixpoint**(const char * *str*, int *fract_mult*, int * *integer*, int * *fract*)
Parse a fixed-point number from a string

Parameters

const char * str The string to parse

int fract_mult Multiplier for the first decimal place, should be a power of 10

int * integer The integer part of the number

int * fract The fractional part of the number

Description

Returns 0 on success, or a negative error code if the string could not be parsed.

struct *iio_dev* * **iio_device_alloc**(int *sizeof_priv*)
allocate an iio_dev from a driver

Parameters

int sizeof_priv Space to allocate for private structure.

void **iio_device_free**(struct *iio_dev* * *dev*)
free an iio_dev from a driver

Parameters

struct iio_dev * dev the iio_dev associated with the device

struct *iio_dev* * **devm_iio_device_alloc**(struct *device* * *dev*, int *sizeof_priv*)
Resource-managed `iio_device_alloc()`

Parameters

struct device * dev Device to allocate iio_dev for

int sizeof_priv Space to allocate for private structure.

Description

Managed iio_device_alloc. iio_dev allocated with this function is automatically freed on driver detach.

If an iio_dev allocated with this function needs to be freed separately, `devm_iio_device_free()` must be used.

Return

Pointer to allocated iio_dev on success, NULL on failure.

void **devm_iio_device_free**(struct *device* * *dev*, struct *iio_dev* * *iio_dev*)
Resource-managed `iio_device_free()`

Parameters

struct device * dev Device this iio_dev belongs to

struct iio_dev * iio_dev the iio_dev associated with the device

Description

Free iio_dev allocated with `devm_iio_device_alloc()`.

void **iio_device_unregister**(struct *iio_dev* * *indio_dev*)
unregister a device from the IIO subsystem

Parameters

struct iio_dev * indio_dev Device structure representing the device.

void **devm_iio_device_unregister**(struct *device* * *dev*, struct *iio_dev* * *indio_dev*)
Resource-managed *iio_device_unregister()*

Parameters

struct device * dev Device this iio_dev belongs to

struct iio_dev * indio_dev the iio_dev associated with the device

Description

Unregister iio_dev registered with *devm_iio_device_register()*.

int **iio_device_claim_direct_mode**(struct *iio_dev* * *indio_dev*)
Keep device in direct mode

Parameters

struct iio_dev * indio_dev the iio_dev associated with the device

Description

If the device is in direct mode it is guaranteed to stay that way until *iio_device_release_direct_mode()* is called.

Use with *iio_device_release_direct_mode()*

Return

0 on success, -EBUSY on failure

void **iio_device_release_direct_mode**(struct *iio_dev* * *indio_dev*)
releases claim on direct mode

Parameters

struct iio_dev * indio_dev the iio_dev associated with the device

Description

Release the claim. Device is no longer guaranteed to stay in direct mode.

Use with *iio_device_claim_direct_mode()*

Buffers

- struct `iio_buffer` — general buffer structure
- *iio_validate_scan_mask_onehot()* — Validates that exactly one channel is selected
- *iio_buffer_get()* — Grab a reference to the buffer
- *iio_buffer_put()* — Release the reference to the buffer

The Industrial I/O core offers a way for continuous data capture based on a trigger source. Multiple data channels can be read at once from `/dev/iio:device`*X* character device node, thus reducing the CPU load.

IIO buffer sysfs interface

An IIO buffer has an associated attributes directory under `/sys/bus/iio/iio:device`*X*`/buffer/*`. Here are some of the existing attributes:

- `length`, the total number of data samples (capacity) that can be stored by the buffer.
- `enable`, activate buffer capture.

IIO buffer setup

The meta information associated with a channel reading placed in a buffer is called a scan element . The important bits configuring scan elements are exposed to userspace applications via the `/sys/bus/iio/iio:deviceX/scan_elements/*` directory. This file contains attributes of the following form:

- `enable`, used for enabling a channel. If and only if its attribute is non *zero*, then a triggered capture will contain data samples for this channel.
- `type`, description of the scan element data storage within the buffer and hence the form in which it is read from user space. Format is [be|le]:[s|u]bits/storagebitsXrepeat[>>shift] . * *be* or *le*, specifies big or little endian. * *s* or *u*, specifies if signed (2's complement) or unsigned. * *bits*, is the number of valid data bits. * *storagebits*, is the number of bits (after padding) that it occupies in the buffer. * *shift*, if specified, is the shift that needs to be applied prior to masking out unused bits. * *repeat*, specifies the number of bits/storagebits repetitions. When the repeat element is 0 or 1, then the repeat value is omitted.

For example, a driver for a 3-axis accelerometer with 12 bit resolution where data is stored in two 8-bits registers as follows:

```
  7   6   5   4   3   2   1   0
+---+---+---+---+---+---+---+---+
|D3 |D2 |D1 |D0 | X | X | X | X | (LOW byte, address 0x06)
+---+---+---+---+---+---+---+---+

  7   6   5   4   3   2   1   0
+---+---+---+---+---+---+---+---+
|D11|D10|D9 |D8 |D7 |D6 |D5 |D4 | (HIGH byte, address 0x07)
+---+---+---+---+---+---+---+---+
```

will have the following scan element type for each axis:

```
$ cat /sys/bus/iio/devices/iio:device0/scan_elements/in_accel_y_type
le:s12/16>>4
```

A user space application will interpret data samples read from the buffer as two byte little endian signed data, that needs a 4 bits right shift before masking out the 12 valid bits of data.

For implementing buffer support a driver should initialize the following fields in iio_chan_spec definition:

```
struct iio_chan_spec {
/* other members */
        int scan_index
        struct {
                char sign;
                u8 realbits;
                u8 storagebits;
                u8 shift;
                u8 repeat;
                enum iio_endian endianness;
               } scan_type;
       };
```

The driver implementing the accelerometer described above will have the following channel definition:

```
struct struct iio_chan_spec accel_channels[] = {
        {
                .type = IIO_ACCEL,
                .modified = 1,
                .channel2 = IIO_MOD_X,
                /* other stuff here */
                .scan_index = 0,
                .scan_type = {
                        .sign = 's',
                        .realbits = 12,
                        .storagebits = 16,
                        .shift = 4,
                        .endianness = IIO_LE,
                },
        }
        /* similar for Y (with channel2 = IIO_MOD_Y, scan_index = 1)
         * and Z (with channel2 = IIO_MOD_Z, scan_index = 2) axis
         */
}
```

Here **scan_index** defines the order in which the enabled channels are placed inside the buffer. Channels with a lower **scan_index** will be placed before channels with a higher index. Each channel needs to have a unique **scan_index**.

Setting **scan_index** to -1 can be used to indicate that the specific channel does not support buffered capture. In this case no entries will be created for the channel in the scan_elements directory.

More details

int **iio_push_to_buffers_with_timestamp**(struct iio_dev * *indio_dev*, void * *data*, int64_t *timestamp*)
push data and timestamp to buffers

Parameters

struct iio_dev * indio_dev iio_dev structure for device.

void * data sample data

int64_t timestamp timestamp for the sample data

Description

Pushes data to the IIO device's buffers. If timestamps are enabled for the device the function will store the supplied timestamp as the last element in the sample data buffer before pushing it to the device buffers. The sample data buffer needs to be large enough to hold the additional timestamp (usually the buffer should be indio->scan_bytes bytes large).

Returns 0 on success, a negative error code otherwise.

void **iio_buffer_set_attrs**(struct iio_buffer * *buffer*, const struct attribute ** *attrs*)
Set buffer specific attributes

Parameters

struct iio_buffer * buffer The buffer for which we are setting attributes

const struct attribute ** attrs Pointer to a null terminated list of pointers to attributes

bool **iio_validate_scan_mask_onehot**(struct iio_dev * *indio_dev*, const unsigned long * *mask*)
Validates that exactly one channel is selected

Parameters

struct iio_dev * indio_dev the iio device

const unsigned long * mask scan mask to be checked

Description

Return true if exactly one bit is set in the scan mask, false otherwise. It can be used for devices where only one channel can be active for sampling at a time.

int **iio_push_to_buffers**(struct iio_dev * *indio_dev*, const void * *data*)
push to a registered buffer.

Parameters

`struct iio_dev * indio_dev` iio_dev structure for device.

`const void * data` Full scan.

struct iio_buffer * **`iio_buffer_get`**(struct iio_buffer * *buffer*)
Grab a reference to the buffer

Parameters

`struct iio_buffer * buffer` The buffer to grab a reference for, may be NULL

Description

Returns the pointer to the buffer that was passed into the function.

void **`iio_buffer_put`**(struct iio_buffer * *buffer*)
Release the reference to the buffer

Parameters

`struct iio_buffer * buffer` The buffer to release the reference for, may be NULL

void **`iio_device_attach_buffer`**(struct *iio_dev* * *indio_dev*, struct iio_buffer * *buffer*)
Attach a buffer to a IIO device

Parameters

`struct iio_dev * indio_dev` The device the buffer should be attached to

`struct iio_buffer * buffer` The buffer to attach to the device

Description

This function attaches a buffer to a IIO device. The buffer stays attached to the device until the device is freed. The function should only be called at most once per device.

Triggers

- struct *`iio_trigger`* — industrial I/O trigger device
- *`devm_iio_trigger_alloc()`* — Resource-managed iio_trigger_alloc
- *`devm_iio_trigger_free()`* — Resource-managed iio_trigger_free
- `devm_iio_trigger_register()` — Resource-managed iio_trigger_register
- *`devm_iio_trigger_unregister()`* — Resource-managed iio_trigger_unregister
- *`iio_trigger_validate_own_device()`* — Check if a trigger and IIO device belong to the same device

In many situations it is useful for a driver to be able to capture data based on some external event (trigger) as opposed to periodically polling for data. An IIO trigger can be provided by a device driver that also has an IIO device based on hardware generated events (e.g. data ready or threshold exceeded) or provided by a separate driver from an independent interrupt source (e.g. GPIO line connected to some external system, timer interrupt or user space writing a specific file in sysfs). A trigger may initiate data capture for a number of sensors and also it may be completely unrelated to the sensor itself.

IIO trigger sysfs interface

There are two locations in sysfs related to triggers:

- `/sys/bus/iio/devices/trigger`*`Y`*`/*`, this file is created once an IIO trigger is registered with the IIO core and corresponds to trigger with index Y. Because triggers can be very different depending on type there are few standard attributes that we can describe here:

  - name, trigger name that can be later used for association with a device.
  - `sampling_frequency`, some timer based triggers use this attribute to specify the frequency for trigger calls.
- `/sys/bus/iio/devices/iio:device`*X*`/trigger/*`, this directory is created once the device supports a triggered buffer. We can associate a trigger with our device by writing the trigger's name in the `current_trigger` file.

IIO trigger setup

Let's see a simple example of how to setup a trigger to be used by a driver:

```
struct iio_trigger_ops trigger_ops = {
    .set_trigger_state = sample_trigger_state,
    .validate_device = sample_validate_device,
}

struct iio_trigger *trig;

/* first, allocate memory for our trigger */
trig = iio_trigger_alloc(dev, "trig-%s-%d", name, idx);

/* setup trigger operations field */
trig->ops = &trigger_ops;

/* now register the trigger with the IIO core */
iio_trigger_register(trig);
```

IIO trigger ops

- struct *`iio_trigger_ops`* — operations structure for an iio_trigger.

Notice that a trigger has a set of operations attached:

- `set_trigger_state`, switch the trigger on/off on demand.
- `validate_device`, function to validate the device when the current trigger gets changed.

More details

struct **`iio_trigger_ops`**
operations structure for an iio_trigger.

Definition

```
struct iio_trigger_ops {
  int (*set_trigger_state)(struct iio_trigger *trig, bool state);
  int (*try_reenable)(struct iio_trigger *trig);
  int (*validate_device)(struct iio_trigger *trig, struct iio_dev *indio_dev);
};
```

Members

`set_trigger_state` switch on/off the trigger on demand

`try_reenable` function to reenable the trigger when the use count is zero (may be NULL)

`validate_device` function to validate the device when the current trigger gets changed.

Description

This is typically static const within a driver and shared by instances of a given device.

struct **iio_trigger**
industrial I/O trigger device

Definition

```
struct iio_trigger {
  const struct iio_trigger_ops    *ops;
  struct module                   *owner;
  int id;
  const char                      *name;
  struct device                   dev;
  struct list_head                list;
  struct list_head                alloc_list;
  atomic_t use_count;
  struct irq_chip                 subirq_chip;
  int subirq_base;
  struct iio_subirq subirqs[CONFIG_IIO_CONSUMERS_PER_TRIGGER];
  unsigned long pool[BITS_TO_LONGS(CONFIG_IIO_CONSUMERS_PER_TRIGGER)];
  struct mutex                    pool_lock;
  bool attached_own_device;
};
```

Members

ops [DRIVER] operations structure

owner [INTERN] owner of this driver module

id [INTERN] unique id number

name [DRIVER] unique name

dev [DRIVER] associated device (if relevant)

list [INTERN] used in maintenance of global trigger list

alloc_list [DRIVER] used for driver specific trigger list

use_count [INTERN] use count for the trigger.

subirq_chip [INTERN] associate 'virtual' irq chip.

subirq_base [INTERN] base number for irqs provided by trigger.

subirqs [INTERN] information about the 'child' irqs.

pool [INTERN] bitmap of irqs currently in use.

pool_lock [INTERN] protection of the irq pool.

attached_own_device [INTERN] if we are using our own device as trigger, i.e. if we registered a poll function to the same device as the one providing the trigger.

void **iio_trigger_set_drvdata**(struct *iio_trigger* * *trig*, void * *data*)
Set trigger driver data

Parameters

struct iio_trigger * trig IIO trigger structure

void * data Driver specific data

Description

Allows to attach an arbitrary pointer to an IIO trigger, which can later be retrieved by *iio_trigger_get_drvdata()*.

void * **iio_trigger_get_drvdata**(struct *iio_trigger* * *trig*)
Get trigger driver data

Parameters

struct iio_trigger * trig IIO trigger structure

Description

Returns the data previously set with *iio_trigger_set_drvdata()*

iio_trigger_register(*trig_info*)
register a trigger with the IIO core

Parameters

trig_info trigger to be registered

void **iio_trigger_unregister**(struct *iio_trigger* * *trig_info*)
unregister a trigger from the core

Parameters

struct iio_trigger * trig_info trigger to be unregistered

int **iio_trigger_set_immutable**(struct *iio_dev* * *indio_dev*, struct *iio_trigger* * *trig*)
set an immutable trigger on destination

Parameters

struct iio_dev * indio_dev IIO device structure containing the device

struct iio_trigger * trig trigger to assign to device

void **iio_trigger_poll**(struct *iio_trigger* * *trig*)
called on a trigger occurring

Parameters

struct iio_trigger * trig trigger which occurred

Description

Typically called in relevant hardware interrupt handler.

bool **iio_trigger_using_own**(struct *iio_dev* * *indio_dev*)
tells us if we use our own HW trigger ourselves

Parameters

struct iio_dev * indio_dev device to check

struct *iio_trigger* * **devm_iio_trigger_alloc**(struct *device* * *dev*, const char * *fmt*, ...)
Resource-managed `iio_trigger_alloc()`

Parameters

struct device * dev Device to allocate iio_trigger for

const char * fmt trigger name format. If it includes format specifiers, the additional arguments following format are formatted and inserted in the resulting string replacing their respective specifiers.

... variable arguments

Description

Managed iio_trigger_alloc. iio_trigger allocated with this function is automatically freed on driver detach.

If an iio_trigger allocated with this function needs to be freed separately, *devm_iio_trigger_free()* must be used.

Return

Pointer to allocated iio_trigger on success, NULL on failure.

void **devm_iio_trigger_free**(struct *device* * *dev*, struct *iio_trigger* * *iio_trig*)
Resource-managed `iio_trigger_free()`

Parameters

struct device * dev Device this iio_dev belongs to

struct iio_trigger * iio_trig the iio_trigger associated with the device

Description

Free iio_trigger allocated with *devm_iio_trigger_alloc()*.

int **__devm_iio_trigger_register**(struct *device* * *dev*, struct *iio_trigger* * *trig_info*, struct module * *this_mod*)

Resource-managed *iio_trigger_register()*

Parameters

struct device * dev device this trigger was allocated for

struct iio_trigger * trig_info trigger to register

struct module * this_mod module registering the trigger

Description

Managed *iio_trigger_register()*. The IIO trigger registered with this function is automatically unregistered on driver detach. This function calls *iio_trigger_register()* internally. Refer to that function for more information.

If an iio_trigger registered with this function needs to be unregistered separately, *devm_iio_trigger_unregister()* must be used.

Return

0 on success, negative error number on failure.

void **devm_iio_trigger_unregister**(struct *device* * *dev*, struct *iio_trigger* * *trig_info*)

Resource-managed *iio_trigger_unregister()*

Parameters

struct device * dev device this iio_trigger belongs to

struct iio_trigger * trig_info the trigger associated with the device

Description

Unregister trigger registered with `devm_iio_trigger_register()`.

int **iio_trigger_validate_own_device**(struct *iio_trigger* * *trig*, struct *iio_dev* * *indio_dev*)

Check if a trigger and IIO device belong to the same device

Parameters

struct iio_trigger * trig The IIO trigger to check

struct iio_dev * indio_dev the IIO device to check

Description

This function can be used as the validate_device callback for triggers that can only be attached to their own device.

Return

0 if both the trigger and the IIO device belong to the same device, -EINVAL otherwise.

Triggered Buffers

Now that we know what buffers and triggers are let's see how they work together.

IIO triggered buffer setup

- *iio_triggered_buffer_setup()* — Setup triggered buffer and pollfunc
- *iio_triggered_buffer_cleanup()* — Free resources allocated by *iio_triggered_buffer_setup()*
- struct *iio_buffer_setup_ops* — buffer setup related callbacks

A typical triggered buffer setup looks like this:

```
const struct iio_buffer_setup_ops sensor_buffer_setup_ops = {
  .preenable    = sensor_buffer_preenable,
  .postenable   = sensor_buffer_postenable,
  .postdisable  = sensor_buffer_postdisable,
  .predisable   = sensor_buffer_predisable,
};

irqreturn_t sensor_iio_pollfunc(int irq, void *p)
{
    pf->timestamp = iio_get_time_ns((struct indio_dev *)p);
    return IRQ_WAKE_THREAD;
}

irqreturn_t sensor_trigger_handler(int irq, void *p)
{
    u16 buf[8];
    int i = 0;

    /* read data for each active channel */
    for_each_set_bit(bit, active_scan_mask, masklength)
        buf[i++] = sensor_get_data(bit)

    iio_push_to_buffers_with_timestamp(indio_dev, buf, timestamp);

    iio_trigger_notify_done(trigger);
    return IRQ_HANDLED;
}

/* setup triggered buffer, usually in probe function */
iio_triggered_buffer_setup(indio_dev, sensor_iio_polfunc,
                           sensor_trigger_handler,
                           sensor_buffer_setup_ops);
```

The important things to notice here are:

- *iio_buffer_setup_ops*, the buffer setup functions to be called at predefined points in the buffer configuration sequence (e.g. before enable, after disable). If not specified, the IIO core uses the default iio_triggered_buffer_setup_ops.
- **sensor_iio_pollfunc**, the function that will be used as top half of poll function. It should do as little processing as possible, because it runs in interrupt context. The most common operation is recording of the current timestamp and for this reason one can use the IIO core defined `iio_pollfunc_store_time()` function.
- **sensor_trigger_handler**, the function that will be used as bottom half of the poll function. This runs in the context of a kernel thread and all the processing takes place here. It usually reads data from the device and stores it in the internal buffer together with the timestamp recorded in the top half.

More details

int **iio_triggered_buffer_setup**(struct *iio_dev* * *indio_dev*, irqreturn_t (*h) (int *irq*, void **p*, irqreturn_t (*thread) (int *irq*, void **p*, const struct *iio_buffer_setup_ops* * *setup_ops*)

Setup triggered buffer and pollfunc

Parameters

struct iio_dev * indio_dev IIO device structure

irqreturn_t (*)(int irq, void *p) h Function which will be used as pollfunc top half

irqreturn_t (*)(int irq, void *p) thread Function which will be used as pollfunc bottom half

const struct iio_buffer_setup_ops * setup_ops Buffer setup functions to use for this device. If NULL the default setup functions for triggered buffers will be used.

Description

This function combines some common tasks which will normally be performed when setting up a triggered buffer. It will allocate the buffer and the pollfunc.

Before calling this function the indio_dev structure should already be completely initialized, but not yet registered. In practice this means that this function should be called right before *iio_device_register()*.

To free the resources allocated by this function call *iio_triggered_buffer_cleanup()*.

void **iio_triggered_buffer_cleanup**(struct *iio_dev* * *indio_dev*)

Free resources allocated by *iio_triggered_buffer_setup()*

Parameters

struct iio_dev * indio_dev IIO device structure

HW consumer

An IIO device can be directly connected to another device in hardware. in this case the buffers between IIO provider and IIO consumer are handled by hardware. The Industrial I/O HW consumer offers a way to bond these IIO devices without software buffer for data. The implementation can be found under `drivers/iio/buffer/hw-consumer.c`

- struct iio_hw_consumer — Hardware consumer structure
- *iio_hw_consumer_alloc()* — Allocate IIO hardware consumer
- *iio_hw_consumer_free()* — Free IIO hardware consumer
- *iio_hw_consumer_enable()* — Enable IIO hardware consumer
- *iio_hw_consumer_disable()* — Disable IIO hardware consumer

HW consumer setup

As standard IIO device the implementation is based on IIO provider/consumer. A typical IIO HW consumer setup looks like this:

```
static struct iio_hw_consumer *hwc;

static const struct iio_info adc_info = {
        .read_raw = adc_read_raw,
};

static int adc_read_raw(struct iio_dev *indio_dev,
                        struct iio_chan_spec const *chan, int *val,
                        int *val2, long mask)
{
        ret = iio_hw_consumer_enable(hwc);

        /* Acquire data */

        ret = iio_hw_consumer_disable(hwc);
}

static int adc_probe(struct platform_device *pdev)
{
        hwc = devm_iio_hw_consumer_alloc(&iio->dev);
}
```

More details

struct iio_hw_consumer * **iio_hw_consumer_alloc**(struct *device* * *dev*)
Allocate IIO hardware consumer

Parameters

struct device * dev Pointer to consumer device.

Description

Returns a valid iio_hw_consumer on success or a ERR_PTR() on failure.

void **iio_hw_consumer_free**(struct iio_hw_consumer * *hwc*)
Free IIO hardware consumer

Parameters

struct iio_hw_consumer * hwc hw consumer to free.

struct iio_hw_consumer * **devm_iio_hw_consumer_alloc**(struct *device* * *dev*)
Resource-managed *iio_hw_consumer_alloc()*

Parameters

struct device * dev Pointer to consumer device.

Description

Managed iio_hw_consumer_alloc. iio_hw_consumer allocated with this function is automatically freed on driver detach.

If an iio_hw_consumer allocated with this function needs to be freed separately, *devm_iio_hw_consumer_free()* must be used.

returns pointer to allocated iio_hw_consumer on success, NULL on failure.

void **devm_iio_hw_consumer_free**(struct *device* * *dev*, struct iio_hw_consumer * *hwc*)
Resource-managed *iio_hw_consumer_free()*

Parameters

struct device * dev Pointer to consumer device.

struct iio_hw_consumer * hwc iio_hw_consumer to free.

Description

Free iio_hw_consumer allocated with *devm_iio_hw_consumer_alloc()*.

int **iio_hw_consumer_enable**(struct iio_hw_consumer * *hwc*)
Enable IIO hardware consumer

Parameters

struct iio_hw_consumer * hwc iio_hw_consumer to enable.

Description

Returns 0 on success.

void **iio_hw_consumer_disable**(struct iio_hw_consumer * *hwc*)
Disable IIO hardware consumer

Parameters

struct iio_hw_consumer * hwc iio_hw_consumer to disable.

CHAPTER

TWELVE

INPUT SUBSYSTEM

Input core

struct **input_value**
input value representation

Definition

```
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

struct **input_dev**
represents an input device

Definition

```
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 *old_keycode
  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 bitmap of device properties and quirks

evbit bitmap of types of events supported by the device (EV_KEY, EV_REL, etc.)

keybit bitmap of keys/buttons this device has

relbit bitmap of relative axes for the device

absbit bitmap of absolute axes for the device

mscbit bitmap of miscellaneous events supported by the device

ledbit bitmap of leds present on the device

sndbit bitmap of sound effects supported by the device

ffbit bitmap of force feedback effects supported by the device

swbit 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 current values for autorepeat parameters (delay, rate)

mt pointer to multitouch state

absinfo array of `struct input_absinfo` elements holding information about absolute axes (current value, min, max, flat, fuzz, resolution)

key reflects current state of device's keys/buttons

led reflects current state of device's LEDs

snd reflects current state of sound effects

sw reflects current state of device's switches

open this method is called when the very first user calls *input_open_device()*. 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 *input_close_device()*.

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 _to_ the device, like EV_LED or EV_SND. The device is expected to carry out the requested action (turn on a LED, play sound, etc.) The call is protected by **event_lock** and must not sleep

grab input handle that currently has the device grabbed (via EVIOCGRAB ioctl). When a handle grabs a device it becomes sole recipient for all input events coming from the device

event_lock this spinlock is taken when input core receives and processes a new event for the device (in *input_event()*). 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 `open()`, `close()` and `flush()` methods

users stores number of users (input handlers) that opened this device. It is used by *input_open_device()* and *input_close_device()* to make sure that dev->:c:func:*open()* is only called when the first user opens device and dev->:c:func:*close()* is called when the very last user closes the device

going_away marks devices that are in a middle of unregistering and causes input_open_device*() fail with -ENODEV.

dev driver model's view of this device

h_list list of input handles associated with the device. When accessing the list dev->mutex 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.

struct **input_handler**
implements one of interfaces for input devices

Definition

```
struct input_handler {
  void *private;
  void (*event)(struct input_handle *handle, unsigned int type, unsigned int code, int value);
  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, int value);
  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 **event**; 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.

struct **`input_handle`**
links input device with an input handler

Definition

```
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

void **input_set_events_per_packet**(struct *input_dev* * *dev*, int *n_events*)
tell handlers about the driver event rate

Parameters

struct input_dev * dev the input device used by the driver

int 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.

struct **ff_device**
force-feedback part of an input device

Definition

```
struct ff_device {
  int (*upload)(struct input_dev *dev, struct ff_effect *effect, struct ff_effect *old);
  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 ffbit[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

`ffbit` bitmap of force feedback capabilities truly supported by device (not emulated like ones in input_dev->ffbit)

`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.

void **`input_event`**(struct *input_dev* * *dev*, unsigned int *type*, unsigned int *code*, int *value*)
report new input event

Parameters

`struct input_dev * dev` device that generated the event

`unsigned int type` type of the event

`unsigned int code` event code

`int 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.

void **`input_inject_event`**(struct *input_handle* * *handle*, unsigned int *type*, unsigned int *code*, int *value*)
send input event from input handler

Parameters

`struct input_handle * handle` input handle to send event through

`unsigned int type` type of the event

`unsigned int code` event code

`int 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.

void **`input_alloc_absinfo`**(struct *input_dev* * *dev*)
allocates array of input_absinfo structs

Parameters

`struct input_dev * 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.

int **`input_grab_device`**(struct *input_handle* * *handle*)
grabs device for exclusive use

Parameters

`struct input_handle * 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.

void **`input_release_device`**(struct *input_handle* * *handle*)
release previously grabbed device

Parameters

`struct input_handle * 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.

int **`input_open_device`**(struct *input_handle* * *handle*)
open input device

Parameters

`struct input_handle * 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.

void **`input_close_device`**(struct *input_handle* * *handle*)
close input device

Parameters

`struct input_handle * 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.

int **`input_scancode_to_scalar`**(const struct input_keymap_entry * *ke*, unsigned int * *scancode*)
converts scancode in `struct input_keymap_entry`

Parameters

`const struct input_keymap_entry * ke` keymap entry containing scancode to be converted.

`unsigned int * 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'.

int **input_get_keycode**(struct *input_dev* * *dev*, struct input_keymap_entry * *ke*)
retrieve keycode currently mapped to a given scancode

Parameters

struct input_dev * dev input device which keymap is being queried

struct input_keymap_entry * ke keymap entry

Description

This function should be called by anyone interested in retrieving current keymap. Presently evdev handlers use it.

int **input_set_keycode**(struct *input_dev* * *dev*, const struct input_keymap_entry * *ke*)
attribute a keycode to a given scancode

Parameters

struct input_dev * dev input device which keymap is being updated

const struct input_keymap_entry * ke new keymap entry

Description

This function should be called by anyone needing to update current keymap. Presently keyboard and evdev handlers use it.

void **input_reset_device**(struct *input_dev* * *dev*)
reset/restore the state of input device

Parameters

struct input_dev * 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.

struct *input_dev* * **input_allocate_device**(void)
allocate memory for new input device

Parameters

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.

struct *input_dev* * **devm_input_allocate_device**(struct *device* * *dev*)
allocate managed input device

Parameters

struct device * 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]:c:func:*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.

void **input_free_device**(struct *input_dev* * *dev*)
    free memory occupied by input_dev structure

Parameters

struct input_dev * 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.

void **input_set_capability**(struct *input_dev* * *dev*, unsigned int *type*, unsigned int *code*)
    mark device as capable of a certain event

Parameters

struct input_dev * dev device that is capable of emitting or accepting event

unsigned int type type of the event (EV_KEY, EV_REL, etc...)

unsigned int code event code

Description

In addition to setting up corresponding bit in appropriate capability bitmap the function also adjusts dev->evbit.

void **input_enable_softrepeat**(struct *input_dev* * *dev*, int *delay*, int *period*)
    enable software autorepeat

Parameters

struct input_dev * dev input device

int delay repeat delay

int period repeat period

Description

Enable software autorepeat on the input device.

int **input_register_device**(struct *input_dev* * *dev*)
    register device with input core

Parameters

struct input_dev * dev device to be registered

Description

This function registers device with input core. The device must be allocated with *input_allocate_device()* and all it's 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.

void **`input_unregister_device`**(struct *input_dev* * *dev*)
unregister previously registered device

Parameters

`struct input_dev * 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.

int **`input_register_handler`**(struct *input_handler* * *handler*)
register a new input handler

Parameters

`struct input_handler * 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.

void **`input_unregister_handler`**(struct *input_handler* * *handler*)
unregisters an input handler

Parameters

`struct input_handler * handler` handler to be unregistered

Description

This function disconnects a handler from its input devices and removes it from lists of known handlers.

int **`input_handler_for_each_handle`**(struct *input_handler* * *handler*, void * *data*, int (*fn) (struct *input_handle* *, void *)
handle iterator

Parameters

`struct input_handler * handler` input handler to iterate

`void * data` data for the callback

`int (*)(struct input_handle *, void *) 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.

int **`input_register_handle`**(struct *input_handle* * *handle*)
register a new input handle

Parameters

`struct input_handle * 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.

void **`input_unregister_handle`**(struct *input_handle* * *handle*)
unregister an input handle

Parameters

`struct input_handle * 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.

int **`input_get_new_minor`**(int *legacy_base*, unsigned int *legacy_num*, bool *allow_dynamic*)
allocates a new input minor number

Parameters

`int legacy_base` beginning or the legacy range to be searched

`unsigned int legacy_num` size of legacy range

`bool 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.

void **`input_free_minor`**(unsigned int *minor*)
release previously allocated minor

Parameters

`unsigned int minor` minor to be released

Description

This function releases previously allocated input minor so that it can be reused later.

int **`input_ff_upload`**(struct *input_dev* * *dev*, struct ff_effect * *effect*, struct file * *file*)
upload effect into force-feedback device

Parameters

`struct input_dev * dev` input device

`struct ff_effect * effect` effect to be uploaded

`struct file * file` owner of the effect

int **`input_ff_erase`**(struct *input_dev* * *dev*, int *effect_id*, struct file * *file*)
erase a force-feedback effect from device

Parameters

`struct input_dev * dev` input device to erase effect from

`int effect_id` id of the effect to be erased

`struct file * 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.

int **`input_ff_event`**(struct *input_dev* * *dev*, unsigned int *type*, unsigned int *code*, int *value*)
generic handler for force-feedback events

Parameters

`struct input_dev * dev` input device to send the effect to

`unsigned int type` event type (anything but EV_FF is ignored)

`unsigned int code` event code

int value event value

int **input_ff_create**(struct *input_dev* * *dev*, unsigned int *max_effects*)
create force-feedback device

Parameters

struct input_dev * dev input device supporting force-feedback

unsigned int 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

void **input_ff_destroy**(struct *input_dev* * *dev*)
frees force feedback portion of input device

Parameters

struct input_dev * 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.

int **input_ff_create_memless**(struct *input_dev* * *dev*, void * *data*, int (*play_effect) (struct *input_dev* *, void *, struct ff_effect *))
create memoryless force-feedback device

Parameters

struct input_dev * dev input device supporting force-feedback

void * data driver-specific data to be passed into **play_effect**

int (*)(struct input_dev *, void *, struct ff_effect *) play_effect driver-specific method for playing FF effect

Multitouch Library

struct **input_mt_slot**
represents the state of an input MT slot

Definition

```
struct input_mt_slot {
  int abs[ABS_MT_LAST - ABS_MT_FIRST + 1];
  unsigned int frame;
  unsigned int key;
};
```

Members

abs 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

struct **input_mt**
state of tracked contacts

Definition

```
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 *`input_mt_sync_frame()`* is called

`red` reduced cost matrix for in-kernel tracking

`slots` array of slots holding current values of tracked contacts

struct **`input_mt_pos`**
contact position

Definition

```
struct input_mt_pos {
  s16 x, y;
};
```

Members

`x` horizontal coordinate

`y` vertical coordinate

int **`input_mt_init_slots`**(struct *input_dev* * *dev*, unsigned int *num_slots*, unsigned int *flags*)
initialize MT input slots

Parameters

`struct input_dev * dev` input device supporting MT events and finger tracking

`unsigned int num_slots` number of slots used by the device

`unsigned int 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.

void **`input_mt_destroy_slots`**(struct *input_dev* * *dev*)
frees the MT slots of the input device

Parameters

`struct input_dev * 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.

void **input_mt_report_slot_state**(struct *input_dev* * *dev*, unsigned int *tool_type*, bool *active*)
report contact state

Parameters

struct input_dev * dev input device with allocated MT slots

unsigned int tool_type the tool type to use in this slot

bool active 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.

void **input_mt_report_finger_count**(struct *input_dev* * *dev*, int *count*)
report contact count

Parameters

struct input_dev * dev input device with allocated MT slots

int 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.

void **input_mt_report_pointer_emulation**(struct *input_dev* * *dev*, bool *use_count*)
common pointer emulation

Parameters

struct input_dev * dev input device with allocated MT slots

bool 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.

void **input_mt_drop_unused**(struct *input_dev* * *dev*)
Inactivate slots not seen in this frame

Parameters

struct input_dev * dev input device with allocated MT slots

Description

Lift all slots not seen since the last call to this function.

void **input_mt_sync_frame**(struct *input_dev* * *dev*)
synchronize mt frame

Parameters

struct input_dev * 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.

int **input_mt_assign_slots**(struct *input_dev* * *dev*, int * *slots*, const struct *input_mt_pos* * *pos*, int *num_pos*, int *dmax*)
perform a best-match assignment

Parameters

struct input_dev * dev input device with allocated MT slots

int * slots the slot assignment to be filled

const struct input_mt_pos * pos the position array to match

int num_pos number of positions

int dmax 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.

int **input_mt_get_slot_by_key**(struct *input_dev* * *dev*, int *key*)
return slot matching key

Parameters

struct input_dev * dev input device with allocated MT slots

int 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

struct **input_polled_dev**
simple polled input device

Definition

```
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 `poll()` 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 **poll_interval**.

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.

struct *input_polled_dev* * **input_allocate_polled_device**(void)
allocate memory for polled device

Parameters

void no arguments

Description

The function allocates memory for a polled device and also for an input device associated with this polled device.

struct *input_polled_dev* * **devm_input_allocate_polled_device**(struct *device* * *dev*)
allocate managed polled device

Parameters

struct device * 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.

void **input_free_polled_device**(struct *input_polled_dev* * *dev*)
free memory allocated for polled device

Parameters

struct input_polled_dev * dev device to free

Description

The function frees memory allocated for polling device and drops reference to the associated input device.

int **input_register_polled_device**(struct *input_polled_dev* * *dev*)
register polled device

Parameters

struct input_polled_dev * 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.

void **input_unregister_polled_device**(struct *input_polled_dev* * *dev*)
unregister polled device

Parameters

struct input_polled_dev * 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

struct **matrix_keymap_data**
keymap for matrix keyboards

Definition

```
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.

struct **matrix_keypad_platform_data**
platform-dependent keypad data

Definition

```
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;
  bool drive_inactive_cols;
};
```

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 reporesenting colums

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

drive_inactive_cols drive inactive columns during scan, rather than making them inputs.

Description

This structure represents platform-specific data that use used by matrix_keypad driver to perform proper initialization.

Sparse keymap support

struct **key_entry**
keymap entry for use in sparse keymap

Definition

```
struct key_entry {
  int type;
  u32 code;
  union {
    u16 keycode;
    struct {
      u8 code;
      u8 value;
    } sw;
  };
};
```

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

keycode KEY_* code assigned to a key/button

sw.code SW_* code assigned to a switch

sw.value Value that should be sent in an input even when KE_SW switch is toggled. KE_VSW switches ignore this field and expect driver to supply value for the event.

Description

This structure defines an entry in a sparse keymap used by some input devices for which traditional table-based approach is not suitable.

struct *key_entry* * **sparse_keymap_entry_from_scancode**(struct *input_dev* * *dev*, unsigned int *code*)

perform sparse keymap lookup

Parameters

struct input_dev * dev Input device using sparse keymap

unsigned int code Scan code

Description

This function is used to perform *struct key_entry* lookup in an input device using sparse keymap.

struct *key_entry* * **sparse_keymap_entry_from_keycode**(struct *input_dev* * *dev*, unsigned int *keycode*)

perform sparse keymap lookup

Parameters

struct input_dev * dev Input device using sparse keymap

unsigned int keycode Key code

Description

This function is used to perform *struct key_entry* lookup in an input device using sparse keymap.

int **sparse_keymap_setup**(struct *input_dev* * *dev*, const struct *key_entry* * *keymap*, int (*setup)(struct *input_dev* *, struct *key_entry* *))

set up sparse keymap for an input device

Parameters

struct input_dev * dev Input device

const struct key_entry * keymap Keymap in form of array of *key_entry* structures ending with KE_END type entry

int (*)(struct input_dev *, struct key_entry *) setup Function that can be used to adjust keymap entries depending on device's needs, may be NULL

Description

The function calculates size and allocates copy of the original keymap after which sets up input device event bits appropriately. The allocated copy of the keymap is automatically freed when it is no longer needed.

void **sparse_keymap_report_entry**(struct *input_dev* * *dev*, const struct *key_entry* * *ke*, unsigned int *value*, bool *autorelease*)

report event corresponding to given key entry

Parameters

struct input_dev * dev Input device for which event should be reported

const struct key_entry * ke key entry describing event

unsigned int value Value that should be reported (ignored by KE_SW entries)

bool autorelease 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*.

bool **sparse_keymap_report_event**(struct *input_dev* * *dev*, unsigned int *code*, unsigned int *value*, bool *autorelease*)
report event corresponding to given scancode

Parameters

struct input_dev * dev Input device using sparse keymap

unsigned int code Scan code

unsigned int value Value that should be reported (ignored by KE_SW entries)

bool autorelease 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

THIRTEEN

LINUX USB API

The Linux-USB Host Side API

Introduction to USB on Linux

A Universal Serial Bus (USB) is used to connect a host, such as a PC or workstation, to a number of peripheral devices. USB uses a tree structure, with the host as the root (the system's master), hubs as interior nodes, and peripherals as leaves (and slaves). Modern PCs support several such trees of USB devices, usually a few USB 3.0 (5 GBit/s) or USB 3.1 (10 GBit/s) and some legacy USB 2.0 (480 MBit/s) busses just in case.

That master/slave asymmetry was designed-in for a number of reasons, one being ease of use. It is not physically possible to mistake upstream and downstream or it does not matter with a type C plug (or they are built into the peripheral). Also, the host software doesn't need to deal with distributed auto-configuration since the pre-designated master node manages all that.

Kernel developers added USB support to Linux early in the 2.2 kernel series and have been developing it further since then. Besides support for each new generation of USB, various host controllers gained support, new drivers for peripherals have been added and advanced features for latency measurement and improved power management introduced.

Linux can run inside USB devices as well as on the hosts that control the devices. But USB device drivers running inside those peripherals don't do the same things as the ones running inside hosts, so they've been given a different name: *gadget drivers*. This document does not cover gadget drivers.

USB Host-Side API Model

Host-side drivers for USB devices talk to the "usbcore" APIs. There are two. One is intended for *general-purpose* drivers (exposed through driver frameworks), and the other is for drivers that are *part of the core*. Such core drivers include the *hub* driver (which manages trees of USB devices) and several different kinds of *host controller drivers*, which control individual busses.

The device model seen by USB drivers is relatively complex.

- USB supports four kinds of data transfers (control, bulk, interrupt, and isochronous). Two of them (control and bulk) use bandwidth as it's available, while the other two (interrupt and isochronous) are scheduled to provide guaranteed bandwidth.
- The device description model includes one or more "configurations" per device, only one of which is active at a time. Devices are supposed to be capable of operating at lower than their top speeds and may provide a BOS descriptor showing the lowest speed they remain fully operational at.
- From USB 3.0 on configurations have one or more "functions", which provide a common functionality and are grouped together for purposes of power management.
- Configurations or functions have one or more "interfaces", each of which may have "alternate settings". Interfaces may be standardized by USB "Class" specifications, or may be specific to a vendor or device.

USB device drivers actually bind to interfaces, not devices. Think of them as "interface drivers", though you may not see many devices where the distinction is important. *Most USB devices are simple, with only one function, one configuration, one interface, and one alternate setting.*

- Interfaces have one or more "endpoints", each of which supports one type and direction of data transfer such as "bulk out" or "interrupt in". The entire configuration may have up to sixteen endpoints in each direction, allocated as needed among all the interfaces.
- Data transfer on USB is packetized; each endpoint has a maximum packet size. Drivers must often be aware of conventions such as flagging the end of bulk transfers using "short" (including zero length) packets.
- The Linux USB API supports synchronous calls for control and bulk messages. It also supports asynchronous calls for all kinds of data transfer, using request structures called "URBs" (USB Request Blocks).

Accordingly, the USB Core API exposed to device drivers covers quite a lot of territory. You'll probably need to consult the USB 3.0 specification, available online from www.usb.org at no cost, as well as class or device specifications.

The only host-side drivers that actually touch hardware (reading/writing registers, handling IRQs, and so on) are the HCDs. In theory, all HCDs provide the same functionality through the same API. In practice, that's becoming more true, but there are still differences that crop up especially with fault handling on the less common controllers. Different controllers don't necessarily report the same aspects of failures, and recovery from faults (including software-induced ones like unlinking an URB) isn't yet fully consistent. Device driver authors should make a point of doing disconnect testing (while the device is active) with each different host controller driver, to make sure drivers don't have bugs of their own as well as to make sure they aren't relying on some HCD-specific behavior.

USB-Standard Types

In `<linux/usb/ch9.h>` you will find the USB data types defined in chapter 9 of the USB specification. These data types are used throughout USB, and in APIs including this host side API, gadget APIs, usb character devices and debugfs interfaces.

const char * **`usb_speed_string`**(enum usb_device_speed *speed*)
Returns human readable-name of the speed.

Parameters

`enum usb_device_speed speed` The speed to return human-readable name for. If it's not any of the speeds defined in usb_device_speed enum, string for USB_SPEED_UNKNOWN will be returned.

enum usb_device_speed **`usb_get_maximum_speed`**(struct *device* * *dev*)
Get maximum requested speed for a given USB controller.

Parameters

`struct device * dev` Pointer to the given USB controller device

Description

The function gets the maximum speed string from property "maximum-speed", and returns the corresponding enum usb_device_speed.

const char * **`usb_state_string`**(enum usb_device_state *state*)
Returns human readable name for the state.

Parameters

`enum usb_device_state state` The state to return a human-readable name for. If it's not any of the states devices in usb_device_state_string enum, the string UNKNOWN will be returned.

Host-Side Data Types and Macros

The host side API exposes several layers to drivers, some of which are more necessary than others. These support lifecycle models for host side drivers and devices, and support passing buffers through usbcore to some HCD that performs the I/O for the device driver.

struct **usb_host_endpoint**
host-side endpoint descriptor and queue

Definition

```
struct usb_host_endpoint {
  struct usb_endpoint_descriptor          desc;
  struct usb_ss_ep_comp_descriptor        ss_ep_comp;
  struct usb_ssp_isoc_ep_comp_descriptor  ssp_isoc_ep_comp;
  struct list_head                urb_list;
  void *hcpriv;
  struct ep_device                *ep_dev;
  unsigned char *extra;
  int extralen;
  int enabled;
  int streams;
};
```

Members

desc descriptor for this endpoint, wMaxPacketSize in native byteorder

ss_ep_comp SuperSpeed companion descriptor for this endpoint

ssp_isoc_ep_comp SuperSpeedPlus isoc companion descriptor for this endpoint

urb_list urbs queued to this endpoint; maintained by usbcore

hcpriv for use by HCD; typically holds hardware dma queue head (QH) with one or more transfer descriptors (TDs) per urb

ep_dev ep_device for sysfs info

extra descriptors following this endpoint in the configuration

extralen how many bytes of "extra" are valid

enabled URBs may be submitted to this endpoint

streams number of USB-3 streams allocated on the endpoint

Description

USB requests are always queued to a given endpoint, identified by a descriptor within an active interface in a given USB configuration.

struct **usb_interface**
what usb device drivers talk to

Definition

```
struct usb_interface {
  struct usb_host_interface *altsetting;
  struct usb_host_interface *cur_altsetting;
  unsigned num_altsetting;
  struct usb_interface_assoc_descriptor *intf_assoc;
  int minor;
  enum usb_interface_condition condition;
  unsigned sysfs_files_created:1;
  unsigned ep_devs_created:1;
  unsigned unregistering:1;
  unsigned needs_remote_wakeup:1;
  unsigned needs_altsetting0:1;
  unsigned needs_binding:1;
  unsigned resetting_device:1;
  unsigned authorized:1;
  struct device dev;
  struct device *usb_dev;
  atomic_t pm_usage_cnt;
  struct work_struct reset_ws;
};
```

Members

altsetting array of interface structures, one for each alternate setting that may be selected. Each one includes a set of endpoint configurations. They will be in no particular order.

cur_altsetting the current altsetting.

num_altsetting number of altsettings defined.

intf_assoc interface association descriptor

minor the minor number assigned to this interface, if this interface is bound to a driver that uses the USB major number. If this interface does not use the USB major, this field should be unused. The driver should set this value in the `probe()` function of the driver, after it has been assigned a minor number from the USB core by calling *usb_register_dev()*.

condition binding state of the interface: not bound, binding (in `probe()`), bound to a driver, or unbinding (in `disconnect()`)

sysfs_files_created sysfs attributes exist

ep_devs_created endpoint child pseudo-devices exist

unregistering flag set when the interface is being unregistered

needs_remote_wakeup flag set when the driver requires remote-wakeup capability during autosuspend.

needs_altsetting0 flag set when a set-interface request for altsetting 0 has been deferred.

needs_binding flag set when the driver should be re-probed or unbound following a reset or suspend operation it doesn't support.

resetting_device USB core reset the device, so use alt setting 0 as current; needs bandwidth alloc after reset.

authorized This allows to (de)authorize individual interfaces instead a whole device in contrast to the device authorization.

dev driver model's view of this device

usb_dev if an interface is bound to the USB major, this will point to the sysfs representation for that device.

pm_usage_cnt PM usage counter for this interface

reset_ws Used for scheduling resets from atomic context.

Description

USB device drivers attach to interfaces on a physical device. Each interface encapsulates a single high level function, such as feeding an audio stream to a speaker or reporting a change in a volume control. Many USB devices only have one interface. The protocol used to talk to an interface's endpoints can be defined in a usb "class" specification, or by a product's vendor. The (default) control endpoint is part of every interface, but is never listed among the interface's descriptors.

The driver that is bound to the interface can use standard driver model calls such as `dev_get_drvdata()` on the dev member of this structure.

Each interface may have alternate settings. The initial configuration of a device sets altsetting 0, but the device driver can change that setting using *usb_set_interface()*. Alternate settings are often used to control the use of periodic endpoints, such as by having different endpoints use different amounts of reserved USB bandwidth. All standards-conformant USB devices that use isochronous endpoints will use them in non-default settings.

The USB specification says that alternate setting numbers must run from 0 to one less than the total number of alternate settings. But some devices manage to mess this up, and the structures aren't necessarily stored in numerical order anyhow. Use *usb_altnum_to_altsetting()* to look up an alternate setting in the altsetting array based on its number.

struct **usb_interface_cache**
long-term representation of a device interface

Definition

```
struct usb_interface_cache {
  unsigned num_altsetting;
  struct kref ref;
  struct usb_host_interface altsetting[0];
};
```

Members

num_altsetting number of altsettings defined.

ref reference counter.

altsetting variable-length array of interface structures, one for each alternate setting that may be selected. Each one includes a set of endpoint configurations. They will be in no particular order.

Description

These structures persist for the lifetime of a usb_device, unlike struct usb_interface (which persists only as long as its configuration is installed). The altsetting arrays can be accessed through these structures at any time, permitting comparison of configurations and providing support for the /sys/kernel/debug/usb/devices pseudo-file.

struct **usb_host_config**
representation of a device's configuration

Definition

```
struct usb_host_config {
  struct usb_config_descriptor    desc;
  char *string;
  struct usb_interface_assoc_descriptor *intf_assoc[USB_MAXIADS];
  struct usb_interface *interface[USB_MAXINTERFACES];
  struct usb_interface_cache *intf_cache[USB_MAXINTERFACES];
  unsigned char *extra;
  int extralen;
};
```

Members

desc the device's configuration descriptor.

string pointer to the cached version of the iConfiguration string, if present for this configuration.

intf_assoc list of any interface association descriptors in this config

interface array of pointers to usb_interface structures, one for each interface in the configuration. The number of interfaces is stored in desc.bNumInterfaces. These pointers are valid only while the the configuration is active.

intf_cache array of pointers to usb_interface_cache structures, one for each interface in the configuration. These structures exist for the entire life of the device.

extra pointer to buffer containing all extra descriptors associated with this configuration (those preceding the first interface descriptor).

extralen length of the extra descriptors buffer.

Description

USB devices may have multiple configurations, but only one can be active at any time. Each encapsulates a different operational environment; for example, a dual-speed device would have separate configurations for full-speed and high-speed operation. The number of configurations available is stored in the device descriptor as bNumConfigurations.

A configuration can contain multiple interfaces. Each corresponds to a different function of the USB device, and all are available whenever the configuration is active. The USB standard says that interfaces are supposed to be numbered from 0 to desc.bNumInterfaces-1, but a lot of devices get this wrong. In addition, the interface array is not guaranteed to be sorted in numerical order. Use *usb_ifnum_to_if()* to look up an interface entry based on its number.

Device drivers should not attempt to activate configurations. The choice of which configuration to install is a policy decision based on such considerations as available power, functionality provided, and the user's desires (expressed through userspace tools). However, drivers can call *usb_reset_configuration()* to reinitialize the current configuration and all its interfaces.

struct **usb_device**
kernel's representation of a USB device

Definition

```
struct usb_device {
  int devnum;
  char devpath[16];
  u32 route;
  enum usb_device_state   state;
  enum usb_device_speed   speed;
  struct usb_tt   *tt;
  int ttport;
  unsigned int toggle[2];
  struct usb_device *parent;
  struct usb_bus *bus;
  struct usb_host_endpoint ep0;
  struct device dev;
  struct usb_device_descriptor descriptor;
  struct usb_host_bos *bos;
  struct usb_host_config *config;
  struct usb_host_config *actconfig;
  struct usb_host_endpoint *ep_in[16];
  struct usb_host_endpoint *ep_out[16];
  char **rawdescriptors;
  unsigned short bus_mA;
  u8 portnum;
  u8 level;
  unsigned can_submit:1;
  unsigned persist_enabled:1;
  unsigned have_langid:1;
  unsigned authorized:1;
  unsigned authenticated:1;
  unsigned wusb:1;
  unsigned lpm_capable:1;
  unsigned usb2_hw_lpm_capable:1;
  unsigned usb2_hw_lpm_besl_capable:1;
  unsigned usb2_hw_lpm_enabled:1;
  unsigned usb2_hw_lpm_allowed:1;
  unsigned usb3_lpm_u1_enabled:1;
  unsigned usb3_lpm_u2_enabled:1;
  int string_langid;
  char *product;
  char *manufacturer;
  char *serial;
  struct list_head filelist;
  int maxchild;
  u32 quirks;
  atomic_t urbnum;
  unsigned long active_duration;
#ifdef CONFIG_PM;
  unsigned long connect_time;
  unsigned do_remote_wakeup:1;
  unsigned reset_resume:1;
  unsigned port_is_suspended:1;
#endif;
  struct wusb_dev *wusb_dev;
  int slot_id;
  enum usb_device_removable removable;
  struct usb2_lpm_parameters l1_params;
  struct usb3_lpm_parameters u1_params;
  struct usb3_lpm_parameters u2_params;
  unsigned lpm_disable_count;
  u16 hub_delay;
};
```

Members

devnum device number; address on a USB bus

devpath device ID string for use in messages (e.g., /port/...)

route tree topology hex string for use with xHCI

state device state: configured, not attached, etc.

speed device speed: high/full/low (or error)

tt Transaction Translator info; used with low/full speed dev, highspeed hub

ttport device port on that tt hub

toggle one bit for each endpoint, with ([0] = IN, [1] = OUT) endpoints

parent our hub, unless we're the root

bus bus we're part of

ep0 endpoint 0 data (default control pipe)

dev generic device interface

descriptor USB device descriptor

bos USB device BOS descriptor set

config all of the device's configs

actconfig the active configuration

ep_in array of IN endpoints

ep_out array of OUT endpoints

rawdescriptors raw descriptors for each config

bus_mA Current available from the bus

portnum parent port number (origin 1)

level number of USB hub ancestors

can_submit URBs may be submitted

persist_enabled USB_PERSIST enabled for this device

have_langid whether string_langid is valid

authorized policy has said we can use it; (user space) policy determines if we authorize this device to be used or not. By default, wired USB devices are authorized. WUSB devices are not, until we authorize them from user space. FIXME – complete doc

authenticated Crypto authentication passed

wusb device is Wireless USB

lpm_capable device supports LPM

usb2_hw_lpm_capable device can perform USB2 hardware LPM

usb2_hw_lpm_besl_capable device can perform USB2 hardware BESL LPM

usb2_hw_lpm_enabled USB2 hardware LPM is enabled

usb2_hw_lpm_allowed Userspace allows USB 2.0 LPM to be enabled

usb3_lpm_u1_enabled USB3 hardware U1 LPM enabled

usb3_lpm_u2_enabled USB3 hardware U2 LPM enabled

string_langid language ID for strings

product iProduct string, if present (static)

manufacturer iManufacturer string, if present (static)

serial iSerialNumber string, if present (static)

filelist usbfs files that are open to this device

maxchild number of ports if hub

quirks quirks of the whole device

urbnum number of URBs submitted for the whole device

active_duration total time device is not suspended

connect_time time device was first connected

do_remote_wakeup remote wakeup should be enabled

reset_resume needs reset instead of resume

port_is_suspended the upstream port is suspended (L2 or U3)

wusb_dev if this is a Wireless USB device, link to the WUSB specific data for the device.

slot_id Slot ID assigned by xHCI

removable Device can be physically removed from this port

l1_params best effor service latency for USB2 L1 LPM state, and L1 timeout.

u1_params exit latencies for USB3 U1 LPM state, and hub-initiated timeout.

u2_params exit latencies for USB3 U2 LPM state, and hub-initiated timeout.

lpm_disable_count Ref count used by `usb_disable_lpm()` and `usb_enable_lpm()` to keep track of the number of functions that require USB 3.0 Link Power Management to be disabled for this usb_device. This count should only be manipulated by those functions, with the bandwidth_mutex is held.

hub_delay cached value consisting of: parent->hub_delay + wHubDelay + tTPTransmissionDelay (40ns)

Description

Will be used as wValue for SetIsochDelay requests.

Notes

Usbcore drivers should not set usbdev->state directly. Instead use `usb_set_device_state()`.

`usb_hub_for_each_child`(*hdev*, *port1*, *child*)
iterate over all child devices on the hub

Parameters

`hdev` USB device belonging to the usb hub

`port1` portnum associated with child device

`child` child device pointer

int **`usb_interface_claimed`**(struct *usb_interface* * *iface*)
returns true iff an interface is claimed

Parameters

`struct usb_interface * iface` the interface being checked

Return

`true` (nonzero) iff the interface is claimed, else `false` (zero).

Note

Callers must own the driver model's usb bus readlock. So driver `probe()` entries don't need extra locking, but other call contexts may need to explicitly claim that lock.

int **`usb_make_path`**(struct *usb_device* * *dev*, char * *buf*, size_t *size*)
returns stable device path in the usb tree

Parameters

`struct usb_device * dev` the device whose path is being constructed

`char * buf` where to put the string

`size_t size` how big is "buf"?

Return

Length of the string (> 0) or negative if size was too small.

Note

This identifier is intended to be "stable", reflecting physical paths in hardware such as physical bus addresses for host controllers or ports on USB hubs. That makes it stay the same until systems are physically reconfigured, by re-cabling a tree of USB devices or by moving USB host controllers. Adding and removing devices, including virtual root hubs in host controller driver modules, does not change these path identifiers; neither does rebooting or re-enumerating. These are more useful identifiers than changeable ("unstable") ones like bus numbers or device addresses.

With a partial exception for devices connected to USB 2.0 root hubs, these identifiers are also predictable. So long as the device tree isn't changed, plugging any USB device into a given hub port always gives it the same path. Because of the use of "companion" controllers, devices connected to ports on USB 2.0 root hubs (EHCI host controllers) will get one path ID if they are high speed, and a different one if they are full or low speed.

`USB_DEVICE`(*vend*, *prod*)
macro used to describe a specific usb device

Parameters

`vend` the 16 bit USB Vendor ID

`prod` the 16 bit USB Product ID

Description

This macro is used to create a struct usb_device_id that matches a specific device.

USB_DEVICE_VER(*vend*, *prod*, *lo*, *hi*)
describe a specific usb device with a version range

Parameters

vend the 16 bit USB Vendor ID

prod the 16 bit USB Product ID

lo the bcdDevice_lo value

hi the bcdDevice_hi value

Description

This macro is used to create a struct usb_device_id that matches a specific device, with a version range.

USB_DEVICE_INTERFACE_CLASS(*vend*, *prod*, *cl*)
describe a usb device with a specific interface class

Parameters

vend the 16 bit USB Vendor ID

prod the 16 bit USB Product ID

cl bInterfaceClass value

Description

This macro is used to create a struct usb_device_id that matches a specific interface class of devices.

USB_DEVICE_INTERFACE_PROTOCOL(*vend*, *prod*, *pr*)
describe a usb device with a specific interface protocol

Parameters

vend the 16 bit USB Vendor ID

prod the 16 bit USB Product ID

pr bInterfaceProtocol value

Description

This macro is used to create a struct usb_device_id that matches a specific interface protocol of devices.

USB_DEVICE_INTERFACE_NUMBER(*vend*, *prod*, *num*)
describe a usb device with a specific interface number

Parameters

vend the 16 bit USB Vendor ID

prod the 16 bit USB Product ID

num bInterfaceNumber value

Description

This macro is used to create a struct usb_device_id that matches a specific interface number of devices.

USB_DEVICE_INFO(*cl*, *sc*, *pr*)
macro used to describe a class of usb devices

Parameters

cl bDeviceClass value

sc bDeviceSubClass value

pr bDeviceProtocol value

Description

This macro is used to create a struct usb_device_id that matches a specific class of devices.

USB_INTERFACE_INFO(*cl*, *sc*, *pr*)
macro used to describe a class of usb interfaces

Parameters

cl bInterfaceClass value

sc bInterfaceSubClass value

pr bInterfaceProtocol value

Description

This macro is used to create a struct usb_device_id that matches a specific class of interfaces.

USB_DEVICE_AND_INTERFACE_INFO(*vend*, *prod*, *cl*, *sc*, *pr*)
describe a specific usb device with a class of usb interfaces

Parameters

vend the 16 bit USB Vendor ID

prod the 16 bit USB Product ID

cl bInterfaceClass value

sc bInterfaceSubClass value

pr bInterfaceProtocol value

Description

This macro is used to create a struct usb_device_id that matches a specific device with a specific class of interfaces.

This is especially useful when explicitly matching devices that have vendor specific bDeviceClass values, but standards-compliant interfaces.

USB_VENDOR_AND_INTERFACE_INFO(*vend*, *cl*, *sc*, *pr*)
describe a specific usb vendor with a class of usb interfaces

Parameters

vend the 16 bit USB Vendor ID

cl bInterfaceClass value

sc bInterfaceSubClass value

pr bInterfaceProtocol value

Description

This macro is used to create a struct usb_device_id that matches a specific vendor with a specific class of interfaces.

This is especially useful when explicitly matching devices that have vendor specific bDeviceClass values, but standards-compliant interfaces.

struct **usbdrv_wrap**
wrapper for driver-model structure

Definition

```
struct usbdrv_wrap {
  struct device_driver driver;
  int for_devices;
};
```

Members

driver The driver-model core driver structure.

for_devices Non-zero for device drivers, 0 for interface drivers.

struct **usb_driver**
identifies USB interface driver to usbcore

Definition

```
struct usb_driver {
  const char *name;
  int (*probe) (struct usb_interface *intf, const struct usb_device_id *id);
  void (*disconnect) (struct usb_interface *intf);
  int (*unlocked_ioctl) (struct usb_interface *intf, unsigned int code, void *buf);
  int (*suspend) (struct usb_interface *intf, pm_message_t message);
  int (*resume) (struct usb_interface *intf);
  int (*reset_resume)(struct usb_interface *intf);
  int (*pre_reset)(struct usb_interface *intf);
  int (*post_reset)(struct usb_interface *intf);
  const struct usb_device_id *id_table;
  struct usb_dynids dynids;
  struct usbdrv_wrap drvwrap;
  unsigned int no_dynamic_id:1;
  unsigned int supports_autosuspend:1;
  unsigned int disable_hub_initiated_lpm:1;
  unsigned int soft_unbind:1;
};
```

Members

name The driver name should be unique among USB drivers, and should normally be the same as the module name.

probe Called to see if the driver is willing to manage a particular interface on a device. If it is, probe returns zero and uses usb_set_intfdata() to associate driver-specific data with the interface. It may also use *usb_set_interface()* to specify the appropriate altsetting. If unwilling to manage the interface, return -ENODEV, if genuine IO errors occurred, an appropriate negative errno value.

disconnect Called when the interface is no longer accessible, usually because its device has been (or is being) disconnected or the driver module is being unloaded.

unlocked_ioctl Used for drivers that want to talk to userspace through the "usbfs" filesystem. This lets devices provide ways to expose information to user space regardless of where they do (or don't) show up otherwise in the filesystem.

suspend Called when the device is going to be suspended by the system either from system sleep or runtime suspend context. The return value will be ignored in system sleep context, so do NOT try to continue using the device if suspend fails in this case. Instead, let the resume or reset-resume routine recover from the failure.

resume Called when the device is being resumed by the system.

reset_resume Called when the suspended device has been reset instead of being resumed.

pre_reset Called by *usb_reset_device()* when the device is about to be reset. This routine must not return until the driver has no active URBs for the device, and no more URBs may be submitted until the post_reset method is called.

post_reset Called by *usb_reset_device()* after the device has been reset

id_table USB drivers use ID table to support hotplugging. Export this with MODULE_DEVICE_TABLE(usb,...). This must be set or your driver's probe function will never get called.

dynids used internally to hold the list of dynamically added device ids for this driver.

drvwrap Driver-model core structure wrapper.

no_dynamic_id if set to 1, the USB core will not allow dynamic ids to be added to this driver by preventing the sysfs file from being created.

supports_autosuspend if set to 0, the USB core will not allow autosuspend for interfaces bound to this driver.

disable_hub_initiated_lpm if set to 1, the USB core will not allow hubs to initiate lower power link state transitions when an idle timeout occurs. Device-initiated USB 3.0 link PM will still be allowed.

soft_unbind if set to 1, the USB core will not kill URBs and disable endpoints before calling the driver's disconnect method.

Description

USB interface drivers must provide a name, `probe()` and `disconnect()` methods, and an id_table. Other driver fields are optional.

The id_table is used in hotplugging. It holds a set of descriptors, and specialized data may be associated with each entry. That table is used by both user and kernel mode hotplugging support.

The `probe()` and `disconnect()` methods are called in a context where they can sleep, but they should avoid abusing the privilege. Most work to connect to a device should be done when the device is opened, and undone at the last close. The disconnect code needs to address concurrency issues with respect to `open()` and `close()` methods, as well as forcing all pending I/O requests to complete (by unlinking them as necessary, and blocking until the unlinks complete).

struct **usb_device_driver**
identifies USB device driver to usbcore

Definition

```
struct usb_device_driver {
  const char *name;
  int (*probe) (struct usb_device *udev);
  void (*disconnect) (struct usb_device *udev);
  int (*suspend) (struct usb_device *udev, pm_message_t message);
  int (*resume) (struct usb_device *udev, pm_message_t message);
  struct usbdrv_wrap drvwrap;
  unsigned int supports_autosuspend:1;
};
```

Members

name The driver name should be unique among USB drivers, and should normally be the same as the module name.

probe Called to see if the driver is willing to manage a particular device. If it is, probe returns zero and uses `dev_set_drvdata()` to associate driver-specific data with the device. If unwilling to manage the device, return a negative errno value.

disconnect Called when the device is no longer accessible, usually because it has been (or is being) disconnected or the driver's module is being unloaded.

suspend Called when the device is going to be suspended by the system.

resume Called when the device is being resumed by the system.

drvwrap Driver-model core structure wrapper.

supports_autosuspend if set to 0, the USB core will not allow autosuspend for devices bound to this driver.

Description

USB drivers must provide all the fields listed above except drvwrap.

struct **usb_class_driver**
identifies a USB driver that wants to use the USB major number

Definition

```
struct usb_class_driver {
  char *name;
  char *(*devnode)(struct device *dev, umode_t *mode);
  const struct file_operations *fops;
  int minor_base;
};
```

Members

name the usb class device name for this driver. Will show up in sysfs.

devnode Callback to provide a naming hint for a possible device node to create.

fops pointer to the struct file_operations of this driver.

minor_base the start of the minor range for this driver.

Description

This structure is used for the *usb_register_dev()* and *usb_deregister_dev()* functions, to consolidate a number of the parameters used for them.

module_usb_driver(*__usb_driver*)
Helper macro for registering a USB driver

Parameters

__usb_driver usb_driver struct

Description

Helper macro for USB 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()*

struct **urb**
USB Request Block

Definition

```
struct urb {
  struct list_head urb_list;
  struct list_head anchor_list;
  struct usb_anchor *anchor;
  struct usb_device *dev;
  struct usb_host_endpoint *ep;
  unsigned int pipe;
  unsigned int stream_id;
  int status;
  unsigned int transfer_flags;
  void *transfer_buffer;
  dma_addr_t transfer_dma;
  struct scatterlist *sg;
  int num_mapped_sgs;
  int num_sgs;
  u32 transfer_buffer_length;
  u32 actual_length;
  unsigned char *setup_packet;
  dma_addr_t setup_dma;
  int start_frame;
  int number_of_packets;
  int interval;
  int error_count;
  void *context;
  usb_complete_t complete;
  struct usb_iso_packet_descriptor iso_frame_desc[0];
};
```

Members

`urb_list` For use by current owner of the URB.

`anchor_list` membership in the list of an anchor

`anchor` to anchor URBs to a common mooring

`dev` Identifies the USB device to perform the request.

`ep` Points to the endpoint's data structure. Will eventually replace **pipe**.

`pipe` Holds endpoint number, direction, type, and more. Create these values with the eight macros available; usb_{snd,rcv}TYPEpipe(dev,endpoint), where the TYPE is "ctrl" (control), "bulk", "int" (interrupt), or "iso" (isochronous). For example `usb_sndbulkpipe()` or `usb_rcvintpipe()`. Endpoint numbers range from zero to fifteen. Note that "in" endpoint two is a different endpoint (and pipe) from "out" endpoint two. The current configuration controls the existence, type, and maximum packet size of any given endpoint.

`stream_id` the endpoint's stream ID for bulk streams

`status` This is read in non-iso completion functions to get the status of the particular request. ISO requests only use it to tell whether the URB was unlinked; detailed status for each frame is in the fields of the iso_frame-desc.

`transfer_flags` A variety of flags may be used to affect how URB submission, unlinking, or operation are handled. Different kinds of URB can use different flags.

`transfer_buffer` This identifies the buffer to (or from) which the I/O request will be performed unless URB_NO_TRANSFER_DMA_MAP is set (however, do not leave garbage in transfer_buffer even then). This buffer must be suitable for DMA; allocate it with `kmalloc()` or equivalent. For transfers to "in" endpoints, contents of this buffer will be modified. This buffer is used for the data stage of control transfers.

`transfer_dma` When transfer_flags includes URB_NO_TRANSFER_DMA_MAP, the device driver is saying that it provided this DMA address, which the host controller driver should use in preference to the transfer_buffer.

`sg` scatter gather buffer list, the buffer size of each element in the list (except the last) must be divisible by the endpoint's max packet size if no_sg_constraint isn't set in 'struct usb_bus'

`num_mapped_sgs` (internal) number of mapped sg entries

`num_sgs` number of entries in the sg list

`transfer_buffer_length` How big is transfer_buffer. The transfer may be broken up into chunks according to the current maximum packet size for the endpoint, which is a function of the configuration and is encoded in the pipe. When the length is zero, neither transfer_buffer nor transfer_dma is used.

`actual_length` This is read in non-iso completion functions, and it tells how many bytes (out of transfer_buffer_length) were transferred. It will normally be the same as requested, unless either an error was reported or a short read was performed. The URB_SHORT_NOT_OK transfer flag may be used to make such short reads be reported as errors.

`setup_packet` Only used for control transfers, this points to eight bytes of setup data. Control transfers always start by sending this data to the device. Then transfer_buffer is read or written, if needed.

`setup_dma` DMA pointer for the setup packet. The caller must not use this field; setup_packet must point to a valid buffer.

`start_frame` Returns the initial frame for isochronous transfers.

`number_of_packets` Lists the number of ISO transfer buffers.

`interval` Specifies the polling interval for interrupt or isochronous transfers. The units are frames (milliseconds) for full and low speed devices, and microframes (1/8 millisecond) for highspeed and SuperSpeed devices.

`error_count` Returns the number of ISO transfers that reported errors.

context For use in completion functions. This normally points to request-specific driver context.

complete Completion handler. This URB is passed as the parameter to the completion function. The completion function may then do what it likes with the URB, including resubmitting or freeing it.

iso_frame_desc Used to provide arrays of ISO transfer buffers and to collect the transfer status for each buffer.

Description

This structure identifies USB transfer requests. URBs must be allocated by calling *usb_alloc_urb()* and freed with a call to *usb_free_urb()*. Initialization may be done using various usb_fill_*:c:func:_*urb()* functions. URBs are submitted using *usb_submit_urb()*, and pending requests may be canceled using *usb_unlink_urb()* or *usb_kill_urb()*.

Data Transfer Buffers:

Normally drivers provide I/O buffers allocated with `kmalloc()` or otherwise taken from the general page pool. That is provided by transfer_buffer (control requests also use setup_packet), and host controller drivers perform a dma mapping (and unmapping) for each buffer transferred. Those mapping operations can be expensive on some platforms (perhaps using a dma bounce buffer or talking to an IOMMU), although they're cheap on commodity x86 and ppc hardware.

Alternatively, drivers may pass the URB_NO_TRANSFER_DMA_MAP transfer flag, which tells the host controller driver that no such mapping is needed for the transfer_buffer since the device driver is DMA-aware. For example, a device driver might allocate a DMA buffer with *usb_alloc_coherent()* or call *usb_buffer_map()*. When this transfer flag is provided, host controller drivers will attempt to use the dma address found in the transfer_dma field rather than determining a dma address themselves.

Note that transfer_buffer must still be set if the controller does not support DMA (as indicated by bus.uses_dma) and when talking to root hub. If you have to trasfer between highmem zone and the device on such controller, create a bounce buffer or bail out with an error. If transfer_buffer cannot be set (is in highmem) and the controller is DMA capable, assign NULL to it, so that usbmon knows not to use the value. The setup_packet must always be set, so it cannot be located in highmem.

Initialization:

All URBs submitted must initialize the dev, pipe, transfer_flags (may be zero), and complete fields. All URBs must also initialize transfer_buffer and transfer_buffer_length. They may provide the URB_SHORT_NOT_OK transfer flag, indicating that short reads are to be treated as errors; that flag is invalid for write requests.

Bulk URBs may use the URB_ZERO_PACKET transfer flag, indicating that bulk OUT transfers should always terminate with a short packet, even if it means adding an extra zero length packet.

Control URBs must provide a valid pointer in the setup_packet field. Unlike the transfer_buffer, the setup_packet may not be mapped for DMA beforehand.

Interrupt URBs must provide an interval, saying how often (in milliseconds or, for highspeed devices, 125 microsecond units) to poll for transfers. After the URB has been submitted, the interval field reflects how the transfer was actually scheduled. The polling interval may be more frequent than requested. For example, some controllers have a maximum interval of 32 milliseconds, while others support intervals of up to 1024 milliseconds. Isochronous URBs also have transfer intervals. (Note that for isochronous endpoints, as well as high speed interrupt endpoints, the encoding of the transfer interval in the endpoint descriptor is logarithmic. Device drivers must convert that value to linear units themselves.)

If an isochronous endpoint queue isn't already running, the host controller will schedule a new URB to start as soon as bandwidth utilization allows. If the queue is running then a new URB will be scheduled to start in the first transfer slot following the end of the preceding URB, if that slot has not already expired. If the slot has expired (which can happen when IRQ delivery is delayed for a long time), the scheduling behavior depends on the URB_ISO_ASAP flag. If the flag is clear then the URB will be scheduled to start in the expired slot, implying that some of its packets will not be transferred; if the flag is set then the URB will be scheduled in the first unexpired slot, breaking the queue's synchronization. Upon URB completion, the start_frame field will be set to the (micro)frame number in which the transfer was scheduled. Ranges for frame counter values are HC-specific and can go from as low as 256 to as high as 65536 frames.

Isochronous URBs have a different data transfer model, in part because the quality of service is only “best effort”. Callers provide specially allocated URBs, with number_of_packets worth of iso_frame_desc structures at the end. Each such packet is an individual ISO transfer. Isochronous URBs are normally queued, submitted by drivers to arrange that transfers are at least double buffered, and then explicitly resubmitted in completion handlers, so that data (such as audio or video) streams at as constant a rate as the host controller scheduler can support.

Completion Callbacks:

The completion callback is made `in_interrupt()`, and one of the first things that a completion handler should do is check the status field. The status field is provided for all URBs. It is used to report unlinked URBs, and status for all non-ISO transfers. It should not be examined before the URB is returned to the completion handler.

The context field is normally used to link URBs back to the relevant driver or request state.

When the completion callback is invoked for non-isochronous URBs, the actual_length field tells how many bytes were transferred. This field is updated even when the URB terminated with an error or was unlinked.

ISO transfer status is reported in the status and actual_length fields of the iso_frame_desc array, and the number of errors is reported in error_count. Completion callbacks for ISO transfers will normally (re)submit URBs to ensure a constant transfer rate.

Note that even fields marked “public” should not be touched by the driver when the urb is owned by the hcd, that is, since the call to *`usb_submit_urb()`* till the entry into the completion routine.

void **`usb_fill_control_urb`**(struct *urb* * *urb*, struct *usb_device* * *dev*, unsigned int *pipe*, unsigned char * *setup_packet*, void * *transfer_buffer*, int *buffer_length*, usb_complete_t *complete_fn*, void * *context*)

initializes a control urb

Parameters

`struct urb * urb` pointer to the urb to initialize.

`struct usb_device * dev` pointer to the struct usb_device for this urb.

`unsigned int pipe` the endpoint pipe

`unsigned char * setup_packet` pointer to the setup_packet buffer

`void * transfer_buffer` pointer to the transfer buffer

`int buffer_length` length of the transfer buffer

`usb_complete_t complete_fn` pointer to the usb_complete_t function

`void * context` what to set the urb context to.

Description

Initializes a control urb with the proper information needed to submit it to a device.

void **`usb_fill_bulk_urb`**(struct *urb* * *urb*, struct *usb_device* * *dev*, unsigned int *pipe*, void * *transfer_buffer*, int *buffer_length*, usb_complete_t *complete_fn*, void * *context*)

macro to help initialize a bulk urb

Parameters

`struct urb * urb` pointer to the urb to initialize.

`struct usb_device * dev` pointer to the struct usb_device for this urb.

`unsigned int pipe` the endpoint pipe

`void * transfer_buffer` pointer to the transfer buffer

`int buffer_length` length of the transfer buffer

`usb_complete_t complete_fn` pointer to the usb_complete_t function

void * context what to set the urb context to.

Description

Initializes a bulk urb with the proper information needed to submit it to a device.

void **usb_fill_int_urb**(struct *urb* * *urb*, struct *usb_device* * *dev*, unsigned int *pipe*, void * *transfer_buffer*, int *buffer_length*, usb_complete_t *complete_fn*, void * *context*, int *interval*)
macro to help initialize a interrupt urb

Parameters

struct urb * urb pointer to the urb to initialize.

struct usb_device * dev pointer to the struct usb_device for this urb.

unsigned int pipe the endpoint pipe

void * transfer_buffer pointer to the transfer buffer

int buffer_length length of the transfer buffer

usb_complete_t complete_fn pointer to the usb_complete_t function

void * context what to set the urb context to.

int interval what to set the urb interval to, encoded like the endpoint descriptor's bInterval value.

Description

Initializes a interrupt urb with the proper information needed to submit it to a device.

Note that High Speed and SuperSpeed(+) interrupt endpoints use a logarithmic encoding of the endpoint interval, and express polling intervals in microframes (eight per millisecond) rather than in frames (one per millisecond).

Wireless USB also uses the logarithmic encoding, but specifies it in units of 128us instead of 125us. For Wireless USB devices, the interval is passed through to the host controller, rather than being translated into microframe units.

int **usb_urb_dir_in**(struct *urb* * *urb*)
check if an URB describes an IN transfer

Parameters

struct urb * urb URB to be checked

Return

1 if **urb** describes an IN transfer (device-to-host), otherwise 0.

int **usb_urb_dir_out**(struct *urb* * *urb*)
check if an URB describes an OUT transfer

Parameters

struct urb * urb URB to be checked

Return

1 if **urb** describes an OUT transfer (host-to-device), otherwise 0.

struct **usb_sg_request**
support for scatter/gather I/O

Definition

```
struct usb_sg_request {
  int status;
  size_t bytes;
};
```

Members

`status` zero indicates success, else negative errno

`bytes` counts bytes transferred.

Description

These requests are initialized using *`usb_sg_init()`*, and then are used as request handles passed to *`usb_sg_wait()`* or *`usb_sg_cancel()`*. Most members of the request object aren't for driver access.

The status and bytecount values are valid only after *`usb_sg_wait()`* returns. If the status is zero, then the bytecount matches the total from the request.

After an error completion, drivers may need to clear a halt condition on the endpoint.

USB Core APIs

There are two basic I/O models in the USB API. The most elemental one is asynchronous: drivers submit requests in the form of an URB, and the URB's completion callback handles the next step. All USB transfer types support that model, although there are special cases for control URBs (which always have setup and status stages, but may not have a data stage) and isochronous URBs (which allow large packets and include per-packet fault reports). Built on top of that is synchronous API support, where a driver calls a routine that allocates one or more URBs, submits them, and waits until they complete. There are synchronous wrappers for single-buffer control and bulk transfers (which are awkward to use in some driver disconnect scenarios), and for scatterlist based streaming i/o (bulk or interrupt).

USB drivers need to provide buffers that can be used for DMA, although they don't necessarily need to provide the DMA mapping themselves. There are APIs to use used when allocating DMA buffers, which can prevent use of bounce buffers on some systems. In some cases, drivers may be able to rely on 64bit DMA to eliminate another kind of bounce buffer.

void **`usb_init_urb`**(struct *urb* * *urb*)
initializes a urb so that it can be used by a USB driver

Parameters

`struct urb * urb` pointer to the urb to initialize

Description

Initializes a urb so that the USB subsystem can use it properly.

If a urb is created with a call to *`usb_alloc_urb()`* it is not necessary to call this function. Only use this if you allocate the space for a struct urb on your own. If you call this function, be careful when freeing the memory for your urb that it is no longer in use by the USB core.

Only use this function if you _really_ understand what you are doing.

struct *urb* * **`usb_alloc_urb`**(int *iso_packets*, gfp_t *mem_flags*)
creates a new urb for a USB driver to use

Parameters

`int iso_packets` number of iso packets for this urb

`gfp_t mem_flags` the type of memory to allocate, see `kmalloc()` for a list of valid options for this.

Description

Creates an urb for the USB driver to use, initializes a few internal structures, increments the usage counter, and returns a pointer to it.

If the driver want to use this urb for interrupt, control, or bulk endpoints, pass '0' as the number of iso packets.

The driver must call *`usb_free_urb()`* when it is finished with the urb.

Return

A pointer to the new urb, or NULL if no memory is available.

void **usb_free_urb**(struct *urb* * *urb*)
frees the memory used by a urb when all users of it are finished

Parameters

struct urb * urb pointer to the urb to free, may be NULL

Description

Must be called when a user of a urb is finished with it. When the last user of the urb calls this function, the memory of the urb is freed.

Note

The transfer buffer associated with the urb is not freed unless the URB_FREE_BUFFER transfer flag is set.

struct *urb* * **usb_get_urb**(struct *urb* * *urb*)
increments the reference count of the urb

Parameters

struct urb * urb pointer to the urb to modify, may be NULL

Description

This must be called whenever a urb is transferred from a device driver to a host controller driver. This allows proper reference counting to happen for urbs.

Return

A pointer to the urb with the incremented reference counter.

void **usb_anchor_urb**(struct *urb* * *urb*, struct usb_anchor * *anchor*)
anchors an URB while it is processed

Parameters

struct urb * urb pointer to the urb to anchor

struct usb_anchor * anchor pointer to the anchor

Description

This can be called to have access to URBs which are to be executed without bothering to track them

void **usb_unanchor_urb**(struct *urb* * *urb*)
unanchors an URB

Parameters

struct urb * urb pointer to the urb to anchor

Description

Call this to stop the system keeping track of this URB

int **usb_urb_ep_type_check**(const struct *urb* * *urb*)
sanity check of endpoint in the given urb

Parameters

const struct urb * urb urb to be checked

Description

This performs a light-weight sanity check for the endpoint in the given urb. It returns 0 if the urb contains a valid endpoint, otherwise a negative error code.

int **usb_submit_urb**(struct *urb* * *urb*, gfp_t *mem_flags*)
issue an asynchronous transfer request for an endpoint

Parameters

struct urb * urb pointer to the urb describing the request

gfp_t mem_flags the type of memory to allocate, see `kmalloc()` for a list of valid options for this.

Description

This submits a transfer request, and transfers control of the URB describing that request to the USB subsystem. Request completion will be indicated later, asynchronously, by calling the completion handler. The three types of completion are success, error, and unlink (a software-induced fault, also called "request cancellation").

URBs may be submitted in interrupt context.

The caller must have correctly initialized the URB before submitting it. Functions such as *usb_fill_bulk_urb()* and *usb_fill_control_urb()* are available to ensure that most fields are correctly initialized, for the particular kind of transfer, although they will not initialize any transfer flags.

If the submission is successful, the `complete()` callback from the URB will be called exactly once, when the USB core and Host Controller Driver (HCD) are finished with the URB. When the completion function is called, control of the URB is returned to the device driver which issued the request. The completion handler may then immediately free or reuse that URB.

With few exceptions, USB device drivers should never access URB fields provided by usbcore or the HCD until its `complete()` is called. The exceptions relate to periodic transfer scheduling. For both interrupt and isochronous urbs, as part of successful URB submission urb->interval is modified to reflect the actual transfer period used (normally some power of two units). And for isochronous urbs, urb->start_frame is modified to reflect when the URB's transfers were scheduled to start.

Not all isochronous transfer scheduling policies will work, but most host controller drivers should easily handle ISO queues going from now until 10-200 msec into the future. Drivers should try to keep at least one or two msec of data in the queue; many controllers require that new transfers start at least 1 msec in the future when they are added. If the driver is unable to keep up and the queue empties out, the behavior for new submissions is governed by the URB_ISO_ASAP flag. If the flag is set, or if the queue is idle, then the URB is always assigned to the first available (and not yet expired) slot in the endpoint's schedule. If the flag is not set and the queue is active then the URB is always assigned to the next slot in the schedule following the end of the endpoint's previous URB, even if that slot is in the past. When a packet is assigned in this way to a slot that has already expired, the packet is not transmitted and the corresponding usb_iso_packet_descriptor's status field will return -EXDEV. If this would happen to all the packets in the URB, submission fails with a -EXDEV error code.

For control endpoints, the synchronous *usb_control_msg()* call is often used (in non-interrupt context) instead of this call. That is often used through convenience wrappers, for the requests that are standardized in the USB 2.0 specification. For bulk endpoints, a synchronous *usb_bulk_msg()* call is available.

Return

0 on successful submissions. A negative error number otherwise.

Request Queuing:

URBs may be submitted to endpoints before previous ones complete, to minimize the impact of interrupt latencies and system overhead on data throughput. With that queuing policy, an endpoint's queue would never be empty. This is required for continuous isochronous data streams, and may also be required for some kinds of interrupt transfers. Such queuing also maximizes bandwidth utilization by letting USB controllers start work on later requests before driver software has finished the completion processing for earlier (successful) requests.

As of Linux 2.6, all USB endpoint transfer queues support depths greater than one. This was previously a HCD-specific behavior, except for ISO transfers. Non-isochronous endpoint queues are inactive during cleanup after faults (transfer errors or cancellation).

Reserved Bandwidth Transfers:

Periodic transfers (interrupt or isochronous) are performed repeatedly, using the interval specified in the urb. Submitting the first urb to the endpoint reserves the bandwidth necessary to make those transfers.

If the USB subsystem can't allocate sufficient bandwidth to perform the periodic request, submitting such a periodic request should fail.

For devices under xHCI, the bandwidth is reserved at configuration time, or when the alt setting is selected. If there is not enough bus bandwidth, the configuration/alt setting request will fail. Therefore, submissions to periodic endpoints on devices under xHCI should never fail due to bandwidth constraints.

Device drivers must explicitly request that repetition, by ensuring that some URB is always on the endpoint's queue (except possibly for short periods during completion callbacks). When there is no longer an urb queued, the endpoint's bandwidth reservation is canceled. This means drivers can use their completion handlers to ensure they keep bandwidth they need, by reinitializing and resubmitting the just-completed urb until the driver longer needs that periodic bandwidth.

Memory Flags:

The general rules for how to decide which mem_flags to use are the same as for kmalloc. There are four different possible values; GFP_KERNEL, GFP_NOFS, GFP_NOIO and GFP_ATOMIC.

GFP_NOFS is not ever used, as it has not been implemented yet.

GFP_ATOMIC is used when

1. you are inside a completion handler, an interrupt, bottom half, tasklet or timer, or
2. you are holding a spinlock or rwlock (does not apply to semaphores), or
3. current->state != TASK_RUNNING, this is the case only after you've changed it.

GFP_NOIO is used in the block io path and error handling of storage devices.

All other situations use GFP_KERNEL.

Some more specific rules for mem_flags can be inferred, such as

1. start_xmit, timeout, and receive methods of network drivers must use GFP_ATOMIC (they are called with a spinlock held);
2. queuecommand methods of scsi drivers must use GFP_ATOMIC (also called with a spinlock held);
3. If you use a kernel thread with a network driver you must use GFP_NOIO, unless (b) or (c) apply;
4. after you have done a `down()` you can use GFP_KERNEL, unless (b) or (c) apply or your are in a storage driver's block io path;
5. USB probe and disconnect can use GFP_KERNEL unless (b) or (c) apply; and
6. changing firmware on a running storage or net device uses GFP_NOIO, unless b) or c) apply

int **usb_unlink_urb**(struct *urb* * *urb*)
abort/cancel a transfer request for an endpoint

Parameters

struct urb * urb pointer to urb describing a previously submitted request, may be NULL

Description

This routine cancels an in-progress request. URBs complete only once per submission, and may be canceled only once per submission. Successful cancellation means termination of **urb** will be expedited and the completion handler will be called with a status code indicating that the request has been canceled (rather than any other code).

Drivers should not call this routine or related routines, such as `usb_kill_urb()` or `usb_unlink_anchored_urbs()`, after their disconnect method has returned. The disconnect function should synchronize with a driver's I/O routines to insure that all URB-related activity has completed before it returns.

This request is asynchronous, however the HCD might call the ->:c:func:*complete()* callback during unlink. Therefore when drivers call `usb_unlink_urb()`, they must not hold any locks that may be taken by the completion function. Success is indicated by returning -EINPROGRESS, at which time the URB will probably not yet have been given back to the device driver. When it is eventually called, the completion function will see **urb**->status == -ECONNRESET. Failure is indicated by *usb_unlink_urb()* returning any other value. Unlinking will fail when **urb** is not currently "linked" (i.e., it was never submitted, or it was unlinked before, or the hardware is already finished with it), even if the completion handler has not yet run.

The URB must not be deallocated while this routine is running. In particular, when a driver calls this routine, it must insure that the completion handler cannot deallocate the URB.

Return

-EINPROGRESS on success. See description for other values on failure.

Unlinking and Endpoint Queues:

[The behaviors and guarantees described below do not apply to virtual root hubs but only to endpoint queues for physical USB devices.]

Host Controller Drivers (HCDs) place all the URBs for a particular endpoint in a queue. Normally the queue advances as the controller hardware processes each request. But when an URB terminates with an error its queue generally stops (see below), at least until that URB's completion routine returns. It is guaranteed that a stopped queue will not restart until all its unlinked URBs have been fully retired, with their completion routines run, even if that's not until some time after the original completion handler returns. The same behavior and guarantee apply when an URB terminates because it was unlinked.

Bulk and interrupt endpoint queues are guaranteed to stop whenever an URB terminates with any sort of error, including -ECONNRESET, -ENOENT, and -EREMOTEIO. Control endpoint queues behave the same way except that they are not guaranteed to stop for -EREMOTEIO errors. Queues for isochronous endpoints are treated differently, because they must advance at fixed rates. Such queues do not stop when an URB encounters an error or is unlinked. An unlinked isochronous URB may leave a gap in the stream of packets; it is undefined whether such gaps can be filled in.

Note that early termination of an URB because a short packet was received will generate a -EREMOTEIO error if and only if the URB_SHORT_NOT_OK flag is set. By setting this flag, USB device drivers can build deep queues for large or complex bulk transfers and clean them up reliably after any sort of aborted transfer by unlinking all pending URBs at the first fault.

When a control URB terminates with an error other than -EREMOTEIO, it is quite likely that the status stage of the transfer will not take place.

void **usb_kill_urb**(struct *urb* * *urb*)
    cancel a transfer request and wait for it to finish

Parameters

struct urb * urb pointer to URB describing a previously submitted request, may be NULL

Description

This routine cancels an in-progress request. It is guaranteed that upon return all completion handlers will have finished and the URB will be totally idle and available for reuse. These features make this an ideal way to stop I/O in a `disconnect()` callback or `close()` function. If the request has not already finished or been unlinked the completion handler will see urb->status == -ENOENT.

While the routine is running, attempts to resubmit the URB will fail with error -EPERM. Thus even if the URB's completion handler always tries to resubmit, it will not succeed and the URB will become idle.

The URB must not be deallocated while this routine is running. In particular, when a driver calls this routine, it must insure that the completion handler cannot deallocate the URB.

This routine may not be used in an interrupt context (such as a bottom half or a completion handler), or when holding a spinlock, or in other situations where the caller can't `schedule()`.

This routine should not be called by a driver after its disconnect method has returned.

void **usb_poison_urb**(struct *urb* * *urb*)
    reliably kill a transfer and prevent further use of an URB

Parameters

struct urb * urb pointer to URB describing a previously submitted request, may be NULL

Description

This routine cancels an in-progress request. It is guaranteed that upon return all completion handlers will have finished and the URB will be totally idle and cannot be reused. These features make this an ideal way to stop I/O in a `disconnect()` callback. If the request has not already finished or been unlinked the completion handler will see urb->status == -ENOENT.

After and while the routine runs, attempts to resubmit the URB will fail with error -EPERM. Thus even if the URB's completion handler always tries to resubmit, it will not succeed and the URB will become idle.

The URB must not be deallocated while this routine is running. In particular, when a driver calls this routine, it must insure that the completion handler cannot deallocate the URB.

This routine may not be used in an interrupt context (such as a bottom half or a completion handler), or when holding a spinlock, or in other situations where the caller can't `schedule()`.

This routine should not be called by a driver after its disconnect method has returned.

void **`usb_block_urb`**(struct *urb* * *urb*)
reliably prevent further use of an URB

Parameters

`struct urb * urb` pointer to URB to be blocked, may be NULL

Description

After the routine has run, attempts to resubmit the URB will fail with error -EPERM. Thus even if the URB's completion handler always tries to resubmit, it will not succeed and the URB will become idle.

The URB must not be deallocated while this routine is running. In particular, when a driver calls this routine, it must insure that the completion handler cannot deallocate the URB.

void **`usb_kill_anchored_urbs`**(struct usb_anchor * *anchor*)
cancel transfer requests en masse

Parameters

`struct usb_anchor * anchor` anchor the requests are bound to

Description

this allows all outstanding URBs to be killed starting from the back of the queue

This routine should not be called by a driver after its disconnect method has returned.

void **`usb_poison_anchored_urbs`**(struct usb_anchor * *anchor*)
cease all traffic from an anchor

Parameters

`struct usb_anchor * anchor` anchor the requests are bound to

Description

this allows all outstanding URBs to be poisoned starting from the back of the queue. Newly added URBs will also be poisoned

This routine should not be called by a driver after its disconnect method has returned.

void **`usb_unpoison_anchored_urbs`**(struct usb_anchor * *anchor*)
let an anchor be used successfully again

Parameters

`struct usb_anchor * anchor` anchor the requests are bound to

Description

Reverses the effect of usb_poison_anchored_urbs the anchor can be used normally after it returns

void **`usb_unlink_anchored_urbs`**(struct usb_anchor * *anchor*)
asynchronously cancel transfer requests en masse

Parameters

`struct usb_anchor * anchor` anchor the requests are bound to

Description

this allows all outstanding URBs to be unlinked starting from the back of the queue. This function is asynchronous. The unlinking is just triggered. It may happen after this function has returned.

This routine should not be called by a driver after its disconnect method has returned.

void **`usb_anchor_suspend_wakeups`**(struct usb_anchor * *anchor*)

Parameters

`struct usb_anchor * anchor` the anchor you want to suspend wakeups on

Description

Call this to stop the last urb being unanchored from waking up any usb_wait_anchor_empty_timeout waiters. This is used in the hcd urb give- back path to delay waking up until after the completion handler has run.

void **`usb_anchor_resume_wakeups`**(struct usb_anchor * *anchor*)

Parameters

`struct usb_anchor * anchor` the anchor you want to resume wakeups on

Description

Allow usb_wait_anchor_empty_timeout waiters to be woken up again, and wake up any current waiters if the anchor is empty.

int **`usb_wait_anchor_empty_timeout`**(struct usb_anchor * *anchor*, unsigned int *timeout*)
wait for an anchor to be unused

Parameters

`struct usb_anchor * anchor` the anchor you want to become unused

`unsigned int timeout` how long you are willing to wait in milliseconds

Description

Call this is you want to be sure all an anchor's URBs have finished

Return

Non-zero if the anchor became unused. Zero on timeout.

struct *urb* * **`usb_get_from_anchor`**(struct usb_anchor * *anchor*)
get an anchor's oldest urb

Parameters

`struct usb_anchor * anchor` the anchor whose urb you want

Description

This will take the oldest urb from an anchor, unanchor and return it

Return

The oldest urb from **anchor**, or `NULL` if **anchor** has no urbs associated with it.

void **`usb_scuttle_anchored_urbs`**(struct usb_anchor * *anchor*)
unanchor all an anchor's urbs

Parameters

`struct usb_anchor * anchor` the anchor whose urbs you want to unanchor

Description

use this to get rid of all an anchor's urbs

int **usb_anchor_empty**(struct usb_anchor * *anchor*)
is an anchor empty

Parameters

struct usb_anchor * anchor the anchor you want to query

Return

1 if the anchor has no urbs associated with it.

int **usb_control_msg**(struct *usb_device* * *dev*, unsigned int *pipe*, __u8 *request*, __u8 *requesttype*, __u16 *value*, __u16 *index*, void * *data*, __u16 *size*, int *timeout*)
Builds a control urb, sends it off and waits for completion

Parameters

struct usb_device * dev pointer to the usb device to send the message to

unsigned int pipe endpoint "pipe" to send the message to

__u8 request USB message request value

__u8 requesttype USB message request type value

__u16 value USB message value

__u16 index USB message index value

void * data pointer to the data to send

__u16 size length in bytes of the data to send

int timeout time in msecs to wait for the message to complete before timing out (if 0 the wait is forever)

Context

!in_interrupt ()

Description

This function sends a simple control message to a specified endpoint and waits for the message to complete, or timeout.

Don't use this function from within an interrupt context. If you need an asynchronous message, or need to send a message from within interrupt context, use *usb_submit_urb()*. If a thread in your driver uses this call, make sure your `disconnect()` method can wait for it to complete. Since you don't have a handle on the URB used, you can't cancel the request.

Return

If successful, the number of bytes transferred. Otherwise, a negative error number.

int **usb_interrupt_msg**(struct *usb_device* * *usb_dev*, unsigned int *pipe*, void * *data*, int *len*, int * *actual_length*, int *timeout*)
Builds an interrupt urb, sends it off and waits for completion

Parameters

struct usb_device * usb_dev pointer to the usb device to send the message to

unsigned int pipe endpoint "pipe" to send the message to

void * data pointer to the data to send

int len length in bytes of the data to send

int * actual_length pointer to a location to put the actual length transferred in bytes

int timeout time in msecs to wait for the message to complete before timing out (if 0 the wait is forever)

Context

!in_interrupt ()

Description

This function sends a simple interrupt message to a specified endpoint and waits for the message to complete, or timeout.

Don't use this function from within an interrupt context. If you need an asynchronous message, or need to send a message from within interrupt context, use *`usb_submit_urb()`* If a thread in your driver uses this call, make sure your `disconnect()` method can wait for it to complete. Since you don't have a handle on the URB used, you can't cancel the request.

Return

If successful, 0. Otherwise a negative error number. The number of actual bytes transferred will be stored in the **actual_length** parameter.

int **`usb_bulk_msg`**(struct *usb_device* * *usb_dev*, unsigned int *pipe*, void * *data*, int *len*, int * *actual_length*, int *timeout*)
Builds a bulk urb, sends it off and waits for completion

Parameters

`struct usb_device * usb_dev` pointer to the usb device to send the message to

`unsigned int pipe` endpoint "pipe" to send the message to

`void * data` pointer to the data to send

`int len` length in bytes of the data to send

`int * actual_length` pointer to a location to put the actual length transferred in bytes

`int timeout` time in msecs to wait for the message to complete before timing out (if 0 the wait is forever)

Context

!in_interrupt ()

Description

This function sends a simple bulk message to a specified endpoint and waits for the message to complete, or timeout.

Don't use this function from within an interrupt context. If you need an asynchronous message, or need to send a message from within interrupt context, use *`usb_submit_urb()`* If a thread in your driver uses this call, make sure your `disconnect()` method can wait for it to complete. Since you don't have a handle on the URB used, you can't cancel the request.

Because there is no *`usb_interrupt_msg()`* and no USBDEVFS_INTERRUPT ioctl, users are forced to abuse this routine by using it to submit URBs for interrupt endpoints. We will take the liberty of creating an interrupt URB (with the default interval) if the target is an interrupt endpoint.

Return

If successful, 0. Otherwise a negative error number. The number of actual bytes transferred will be stored in the **actual_length** parameter.

int **`usb_sg_init`**(struct *usb_sg_request* * *io*, struct *usb_device* * *dev*, unsigned *pipe*, unsigned *period*, struct scatterlist * *sg*, int *nents*, size_t *length*, gfp_t *mem_flags*)
initializes scatterlist-based bulk/interrupt I/O request

Parameters

`struct usb_sg_request * io` request block being initialized. until *`usb_sg_wait()`* returns, treat this as a pointer to an opaque block of memory,

`struct usb_device * dev` the usb device that will send or receive the data

`unsigned pipe` endpoint "pipe" used to transfer the data

unsigned period polling rate for interrupt endpoints, in frames or (for high speed endpoints) microframes; ignored for bulk

struct scatterlist * sg scatterlist entries

int nents how many entries in the scatterlist

size_t length how many bytes to send from the scatterlist, or zero to send every byte identified in the list.

gfp_t mem_flags SLAB_* flags affecting memory allocations in this call

Description

This initializes a scatter/gather request, allocating resources such as I/O mappings and urb memory (except maybe memory used by USB controller drivers).

The request must be issued using *usb_sg_wait()*, which waits for the I/O to complete (or to be canceled) and then cleans up all resources allocated by *usb_sg_init()*.

The request may be canceled with *usb_sg_cancel()*, either before or after *usb_sg_wait()* is called.

Return

Zero for success, else a negative errno value.

void **usb_sg_wait**(struct *usb_sg_request* * *io*)
synchronously execute scatter/gather request

Parameters

struct usb_sg_request * io request block handle, as initialized with *usb_sg_init()*. some fields become accessible when this call returns.

Context

!in_interrupt ()

Description

This function blocks until the specified I/O operation completes. It leverages the grouping of the related I/O requests to get good transfer rates, by queueing the requests. At higher speeds, such queuing can significantly improve USB throughput.

There are three kinds of completion for this function.

1. success, where io->status is zero. The number of io->bytes transferred is as requested.
2. error, where io->status is a negative errno value. The number of io->bytes transferred before the error is usually less than requested, and can be nonzero.
3. cancellation, a type of error with status -ECONNRESET that is initiated by *usb_sg_cancel()*.

When this function returns, all memory allocated through *usb_sg_init()* or this call will have been freed. The request block parameter may still be passed to *usb_sg_cancel()*, or it may be freed. It could also be reinitialized and then reused.

Data Transfer Rates:

Bulk transfers are valid for full or high speed endpoints. The best full speed data rate is 19 packets of 64 bytes each per frame, or 1216 bytes per millisecond. The best high speed data rate is 13 packets of 512 bytes each per microframe, or 52 KBytes per millisecond.

The reason to use interrupt transfers through this API would most likely be to reserve high speed bandwidth, where up to 24 KBytes per millisecond could be transferred. That capability is less useful for low or full speed interrupt endpoints, which allow at most one packet per millisecond, of at most 8 or 64 bytes (respectively).

It is not necessary to call this function to reserve bandwidth for devices under an xHCI host controller, as the bandwidth is reserved when the configuration or interface alt setting is selected.

void **usb_sg_cancel**(struct *usb_sg_request* * *io*)
stop scatter/gather i/o issued by *usb_sg_wait()*

Parameters

struct usb_sg_request * io request block, initialized with *usb_sg_init()*

Description

This stops a request after it has been started by *usb_sg_wait()*. It can also prevents one initialized by *usb_sg_init()* from starting, so that call just frees resources allocated to the request.

int **usb_get_descriptor**(struct *usb_device* * *dev*, unsigned char *type*, unsigned char *index*, void * *buf*, int *size*)
issues a generic GET_DESCRIPTOR request

Parameters

struct usb_device * dev the device whose descriptor is being retrieved

unsigned char type the descriptor type (USB_DT_*)

unsigned char index the number of the descriptor

void * buf where to put the descriptor

int size how big is "buf"?

Context

!in_interrupt ()

Description

Gets a USB descriptor. Convenience functions exist to simplify getting some types of descriptors. Use usb_get_string() or *usb_string()* for USB_DT_STRING. Device (USB_DT_DEVICE) and configuration descriptors (USB_DT_CONFIG) are part of the device structure. In addition to a number of USB-standard descriptors, some devices also use class-specific or vendor-specific descriptors.

This call is synchronous, and may not be used in an interrupt context.

Return

The number of bytes received on success, or else the status code returned by the underlying *usb_control_msg()* call.

int **usb_string**(struct *usb_device* * *dev*, int *index*, char * *buf*, size_t *size*)
returns UTF-8 version of a string descriptor

Parameters

struct usb_device * dev the device whose string descriptor is being retrieved

int index the number of the descriptor

char * buf where to put the string

size_t size how big is "buf"?

Context

!in_interrupt ()

Description

This converts the UTF-16LE encoded strings returned by devices, from usb_get_string_descriptor(), to null-terminated UTF-8 encoded ones that are more usable in most kernel contexts. Note that this function chooses strings in the first language supported by the device.

This call is synchronous, and may not be used in an interrupt context.

Return

length of the string (>= 0) or usb_control_msg status (< 0).

int **usb_get_status**(struct *usb_device* * *dev*, int *recip*, int *type*, int *target*, void * *data*)
issues a GET_STATUS call

Parameters

struct usb_device * dev the device whose status is being checked

int recip USB_RECIP_*; for device, interface, or endpoint

int type USB_STATUS_TYPE_*; for standard or PTM status types

int target zero (for device), else interface or endpoint number

void * data pointer to two bytes of bitmap data

Context

!in_interrupt ()

Description

Returns device, interface, or endpoint status. Normally only of interest to see if the device is self powered, or has enabled the remote wakeup facility; or whether a bulk or interrupt endpoint is halted ("stalled").

Bits in these status bitmaps are set using the SET_FEATURE request, and cleared using the CLEAR_FEATURE request. The `usb_clear_halt()` function should be used to clear halt ("stall") status.

This call is synchronous, and may not be used in an interrupt context.

Returns 0 and the status value in ***data** (in host byte order) on success, or else the status code from the underlying `usb_control_msg()` call.

int **usb_clear_halt**(struct *usb_device* * *dev*, int *pipe*)
tells device to clear endpoint halt/stall condition

Parameters

struct usb_device * dev device whose endpoint is halted

int pipe endpoint "pipe" being cleared

Context

!in_interrupt ()

Description

This is used to clear halt conditions for bulk and interrupt endpoints, as reported by URB completion status. Endpoints that are halted are sometimes referred to as being "stalled". Such endpoints are unable to transmit or receive data until the halt status is cleared. Any URBs queued for such an endpoint should normally be unlinked by the driver before clearing the halt condition, as described in sections 5.7.5 and 5.8.5 of the USB 2.0 spec.

Note that control and isochronous endpoints don't halt, although control endpoints report "protocol stall" (for unsupported requests) using the same status code used to report a true stall.

This call is synchronous, and may not be used in an interrupt context.

Return

Zero on success, or else the status code returned by the underlying `usb_control_msg()` call.

void **usb_reset_endpoint**(struct *usb_device* * *dev*, unsigned int *epaddr*)
Reset an endpoint's state.

Parameters

struct usb_device * dev the device whose endpoint is to be reset

unsigned int epaddr the endpoint's address. Endpoint number for output, endpoint number + USB_DIR_IN for input

Description

Resets any host-side endpoint state such as the toggle bit, sequence number or current window.

int **usb_set_interface**(struct *usb_device* * *dev*, int *interface*, int *alternate*)
Makes a particular alternate setting be current

Parameters

struct usb_device * dev the device whose interface is being updated

int interface the interface being updated

int alternate the setting being chosen.

Context

!in_interrupt ()

Description

This is used to enable data transfers on interfaces that may not be enabled by default. Not all devices support such configurability. Only the driver bound to an interface may change its setting.

Within any given configuration, each interface may have several alternative settings. These are often used to control levels of bandwidth consumption. For example, the default setting for a high speed interrupt endpoint may not send more than 64 bytes per microframe, while interrupt transfers of up to 3KBytes per microframe are legal. Also, isochronous endpoints may never be part of an interface's default setting. To access such bandwidth, alternate interface settings must be made current.

Note that in the Linux USB subsystem, bandwidth associated with an endpoint in a given alternate setting is not reserved until an URB is submitted that needs that bandwidth. Some other operating systems allocate bandwidth early, when a configuration is chosen.

This call is synchronous, and may not be used in an interrupt context. Also, drivers must not change altsettings while urbs are scheduled for endpoints in that interface; all such urbs must first be completed (perhaps forced by unlinking).

Return

Zero on success, or else the status code returned by the underlying *usb_control_msg()* call.

int **usb_reset_configuration**(struct *usb_device* * *dev*)
lightweight device reset

Parameters

struct usb_device * dev the device whose configuration is being reset

Description

This issues a standard SET_CONFIGURATION request to the device using the current configuration. The effect is to reset most USB-related state in the device, including interface altsettings (reset to zero), endpoint halts (cleared), and endpoint state (only for bulk and interrupt endpoints). Other usbcore state is unchanged, including bindings of usb device drivers to interfaces.

Because this affects multiple interfaces, avoid using this with composite (multi-interface) devices. Instead, the driver for each interface may use *usb_set_interface()* on the interfaces it claims. Be careful though; some devices don't support the SET_INTERFACE request, and others won't reset all the interface state (notably endpoint state). Resetting the whole configuration would affect other drivers' interfaces.

The caller must own the device lock.

Return

Zero on success, else a negative error code.

int **usb_driver_set_configuration**(struct *usb_device* * *udev*, int *config*)
Provide a way for drivers to change device configurations

Parameters

struct usb_device * udev the device whose configuration is being updated

int config the configuration being chosen.

Context

In process context, must be able to sleep

Description

Device interface drivers are not allowed to change device configurations. This is because changing configurations will destroy the interface the driver is bound to and create new ones; it would be like a floppy-disk driver telling the computer to replace the floppy-disk drive with a tape drive!

Still, in certain specialized circumstances the need may arise. This routine gets around the normal restrictions by using a work thread to submit the change-config request.

Return

0 if the request was successfully queued, error code otherwise. The caller has no way to know whether the queued request will eventually succeed.

int **cdc_parse_cdc_header**(struct usb_cdc_parsed_header * *hdr*, struct usb_interface * *intf*, u8 * *buffer*, int *buflen*)
parse the extra headers present in CDC devices

Parameters

struct usb_cdc_parsed_header * hdr the place to put the results of the parsing

struct usb_interface * intf the interface for which parsing is requested

u8 * buffer pointer to the extra headers to be parsed

int buflen length of the extra headers

Description

This evaluates the extra headers present in CDC devices which bind the interfaces for data and control and provide details about the capabilities of the device.

Return

number of descriptors parsed or -EINVAL if the header is contradictory beyond salvage

int **usb_register_dev**(struct usb_interface * *intf*, struct usb_class_driver * *class_driver*)
register a USB device, and ask for a minor number

Parameters

struct usb_interface * intf pointer to the usb_interface that is being registered

struct usb_class_driver * class_driver pointer to the usb_class_driver for this device

Description

This should be called by all USB drivers that use the USB major number. If CONFIG_USB_DYNAMIC_MINORS is enabled, the minor number will be dynamically allocated out of the list of available ones. If it is not enabled, the minor number will be based on the next available free minor, starting at the class_driver->minor_base.

This function also creates a usb class device in the sysfs tree.

usb_deregister_dev() must be called when the driver is done with the minor numbers given out by this function.

Return

-EINVAL if something bad happens with trying to register a device, and 0 on success.

void **usb_deregister_dev**(struct usb_interface * *intf*, struct usb_class_driver * *class_driver*)
deregister a USB device's dynamic minor.

Parameters

struct usb_interface * intf pointer to the usb_interface that is being deregistered

struct usb_class_driver * class_driver pointer to the usb_class_driver for this device

Description

Used in conjunction with *usb_register_dev()*. This function is called when the USB driver is finished with the minor numbers gotten from a call to *usb_register_dev()* (usually when the device is disconnected from the system.)

This function also removes the usb class device from the sysfs tree.

This should be called by all drivers that use the USB major number.

int **usb_driver_claim_interface**(struct *usb_driver* * *driver*, struct *usb_interface* * *iface*, void * *priv*)
bind a driver to an interface

Parameters

struct usb_driver * driver the driver to be bound

struct usb_interface * iface the interface to which it will be bound; must be in the usb device's active configuration

void * priv driver data associated with that interface

Description

This is used by usb device drivers that need to claim more than one interface on a device when probing (audio and acm are current examples). No device driver should directly modify internal usb_interface or usb_device structure members.

Few drivers should need to use this routine, since the most natural way to bind to an interface is to return the private data from the driver's `probe()` method.

Callers must own the device lock, so driver `probe()` entries don't need extra locking, but other call contexts may need to explicitly claim that lock.

Return

0 on success.

void **usb_driver_release_interface**(struct *usb_driver* * *driver*, struct *usb_interface* * *iface*)
unbind a driver from an interface

Parameters

struct usb_driver * driver the driver to be unbound

struct usb_interface * iface the interface from which it will be unbound

Description

This can be used by drivers to release an interface without waiting for their `disconnect()` methods to be called. In typical cases this also causes the driver `disconnect()` method to be called.

This call is synchronous, and may not be used in an interrupt context. Callers must own the device lock, so driver `disconnect()` entries don't need extra locking, but other call contexts may need to explicitly claim that lock.

const struct *usb_device_id* * **usb_match_id**(struct *usb_interface* * *interface*, const struct *usb_device_id* * *id*)
find first usb_device_id matching device or interface

Parameters

struct usb_interface * interface the interface of interest

const struct usb_device_id * id array of usb_device_id structures, terminated by zero entry

Description

usb_match_id searches an array of usb_device_id's and returns the first one matching the device or interface, or null. This is used when binding (or rebinding) a driver to an interface. Most USB device drivers will use this indirectly, through the usb core, but some layered driver frameworks use it directly. These device tables are exported with MODULE_DEVICE_TABLE, through modutils, to support the driver loading functionality of USB hotplugging.

Return

The first matching usb_device_id, or `NULL`.

What Matches:

The "match_flags" element in a usb_device_id controls which members are used. If the corresponding bit is set, the value in the device_id must match its corresponding member in the device or interface descriptor, or else the device_id does not match.

"driver_info" is normally used only by device drivers, but you can create a wildcard "matches anything" usb_device_id as a driver's "modules.usbmap" entry if you provide an id with only a nonzero "driver_info" field. If you do this, the USB device driver's `probe()` routine should use additional intelligence to decide whether to bind to the specified interface.

What Makes Good usb_device_id Tables:

The match algorithm is very simple, so that intelligence in driver selection must come from smart driver id records. Unless you have good reasons to use another selection policy, provide match elements only in related groups, and order match specifiers from specific to general. Use the macros provided for that purpose if you can.

The most specific match specifiers use device descriptor data. These are commonly used with product-specific matches; the USB_DEVICE macro lets you provide vendor and product IDs, and you can also match against ranges of product revisions. These are widely used for devices with application or vendor specific bDeviceClass values.

Matches based on device class/subclass/protocol specifications are slightly more general; use the USB_DEVICE_INFO macro, or its siblings. These are used with single-function devices where bDeviceClass doesn't specify that each interface has its own class.

Matches based on interface class/subclass/protocol are the most general; they let drivers bind to any interface on a multiple-function device. Use the USB_INTERFACE_INFO macro, or its siblings, to match class-per-interface style devices (as recorded in bInterfaceClass).

Note that an entry created by USB_INTERFACE_INFO won't match any interface if the device class is set to Vendor-Specific. This is deliberate; according to the USB spec the meanings of the interface class/subclass/protocol for these devices are also vendor-specific, and hence matching against a standard product class wouldn't work anyway. If you really want to use an interface-based match for such a device, create a match record that also specifies the vendor ID. (Unforunately there isn't a standard macro for creating records like this.)

Within those groups, remember that not all combinations are meaningful. For example, don't give a product version range without vendor and product IDs; or specify a protocol without its associated class and subclass.

int **usb_register_device_driver**(struct *usb_device_driver* * *new_udriver*, struct module * *owner*)
register a USB device (not interface) driver

Parameters

struct usb_device_driver * new_udriver USB operations for the device driver

struct module * owner module owner of this driver.

Description

Registers a USB device driver with the USB core. The list of unattached devices will be rescanned whenever a new driver is added, allowing the new driver to attach to any recognized devices.

Return

A negative error code on failure and 0 on success.

void **usb_deregister_device_driver**(struct *usb_device_driver* * *udriver*)
unregister a USB device (not interface) driver

Parameters

struct usb_device_driver * udriver USB operations of the device driver to unregister

Context

must be able to sleep

Description

Unlinks the specified driver from the internal USB driver list.

int **usb_register_driver**(struct *usb_driver* * *new_driver*, struct module * *owner*, const char * *mod_name*)
register a USB interface driver

Parameters

struct usb_driver * new_driver USB operations for the interface driver

struct module * owner module owner of this driver.

const char * mod_name module name string

Description

Registers a USB interface driver with the USB core. The list of unattached interfaces will be rescanned whenever a new driver is added, allowing the new driver to attach to any recognized interfaces.

Return

A negative error code on failure and 0 on success.

NOTE

if you want your driver to use the USB major number, you must call `usb_register_dev()` to enable that functionality. This function no longer takes care of that.

void **usb_deregister**(struct *usb_driver* * *driver*)
unregister a USB interface driver

Parameters

struct usb_driver * driver USB operations of the interface driver to unregister

Context

must be able to sleep

Description

Unlinks the specified driver from the internal USB driver list.

NOTE

If you called `usb_register_dev()`, you still need to call `usb_deregister_dev()` to clean up your driver's allocated minor numbers, this * call will no longer do it for you.

void **usb_enable_autosuspend**(struct *usb_device* * *udev*)
allow a USB device to be autosuspended

Parameters

struct usb_device * udev the USB device which may be autosuspended

Description

This routine allows **udev** to be autosuspended. An autosuspend won't take place until the autosuspend_delay has elapsed and all the other necessary conditions are satisfied.

The caller must hold **udev**'s device lock.

void **`usb_disable_autosuspend`**(struct *usb_device* * *udev*)
    prevent a USB device from being autosuspended

Parameters

`struct usb_device * udev` the USB device which may not be autosuspended

Description

This routine prevents **udev** from being autosuspended and wakes it up if it is already autosuspended.

The caller must hold **udev**'s device lock.

void **`usb_autopm_put_interface`**(struct *usb_interface* * *intf*)
    decrement a USB interface's PM-usage counter

Parameters

`struct usb_interface * intf` the usb_interface whose counter should be decremented

Description

This routine should be called by an interface driver when it is finished using **intf** and wants to allow it to autosuspend. A typical example would be a character-device driver when its device file is closed.

The routine decrements **intf**'s usage counter. When the counter reaches 0, a delayed autosuspend request for **intf**'s device is attempted. The attempt may fail (see `autosuspend_check()`).

This routine can run only in process context.

void **`usb_autopm_put_interface_async`**(struct *usb_interface* * *intf*)
    decrement a USB interface's PM-usage counter

Parameters

`struct usb_interface * intf` the usb_interface whose counter should be decremented

Description

This routine does much the same thing as *`usb_autopm_put_interface()`*: It decrements **intf**'s usage counter and schedules a delayed autosuspend request if the counter is <= 0. The difference is that it does not perform any synchronization; callers should hold a private lock and handle all synchronization issues themselves.

Typically a driver would call this routine during an URB's completion handler, if no more URBs were pending.

This routine can run in atomic context.

void **`usb_autopm_put_interface_no_suspend`**(struct *usb_interface* * *intf*)
    decrement a USB interface's PM-usage counter

Parameters

`struct usb_interface * intf` the usb_interface whose counter should be decremented

Description

This routine decrements **intf**'s usage counter but does not carry out an autosuspend.

This routine can run in atomic context.

int **`usb_autopm_get_interface`**(struct *usb_interface* * *intf*)
    increment a USB interface's PM-usage counter

Parameters

struct usb_interface * intf the usb_interface whose counter should be incremented

Description

This routine should be called by an interface driver when it wants to use **intf** and needs to guarantee that it is not suspended. In addition, the routine prevents **intf** from being autosuspended subsequently. (Note that this will not prevent suspend events originating in the PM core.) This prevention will persist until *usb_autopm_put_interface()* is called or **intf** is unbound. A typical example would be a character-device driver when its device file is opened.

intf's usage counter is incremented to prevent subsequent autosuspends. However if the autoresume fails then the counter is re-decremented.

This routine can run only in process context.

Return

0 on success.

int **usb_autopm_get_interface_async**(struct *usb_interface* * *intf*)
  increment a USB interface's PM-usage counter

Parameters

struct usb_interface * intf the usb_interface whose counter should be incremented

Description

This routine does much the same thing as *usb_autopm_get_interface()*: It increments **intf**'s usage counter and queues an autoresume request if the device is suspended. The differences are that it does not perform any synchronization (callers should hold a private lock and handle all synchronization issues themselves), and it does not autoresume the device directly (it only queues a request). After a successful call, the device may not yet be resumed.

This routine can run in atomic context.

Return

0 on success. A negative error code otherwise.

void **usb_autopm_get_interface_no_resume**(struct *usb_interface* * *intf*)
  increment a USB interface's PM-usage counter

Parameters

struct usb_interface * intf the usb_interface whose counter should be incremented

Description

This routine increments **intf**'s usage counter but does not carry out an autoresume.

This routine can run in atomic context.

int **usb_find_common_endpoints**(struct usb_host_interface * *alt*, struct usb_endpoint_descriptor ** *bulk_in*, struct usb_endpoint_descriptor ** *bulk_out*, struct usb_endpoint_descriptor ** *int_in*, struct usb_endpoint_descriptor ** *int_out*)

- look up common endpoint descriptors

Parameters

struct usb_host_interface * alt alternate setting to search

struct usb_endpoint_descriptor ** bulk_in pointer to descriptor pointer, or NULL

struct usb_endpoint_descriptor ** bulk_out pointer to descriptor pointer, or NULL

struct usb_endpoint_descriptor ** int_in pointer to descriptor pointer, or NULL

struct usb_endpoint_descriptor ** int_out pointer to descriptor pointer, or NULL

Description

Search the alternate setting's endpoint descriptors for the first bulk-in, bulk-out, interrupt-in and interrupt-out endpoints and return them in the provided pointers (unless they are NULL).

If a requested endpoint is not found, the corresponding pointer is set to NULL.

Return

Zero if all requested descriptors were found, or -ENXIO otherwise.

int **usb_find_common_endpoints_reverse**(struct usb_host_interface * *alt*, struct usb_endpoint_descriptor ** *bulk_in*, struct usb_endpoint_descriptor ** *bulk_out*, struct usb_endpoint_descriptor ** *int_in*, struct usb_endpoint_descriptor ** *int_out*)

- look up common endpoint descriptors

Parameters

struct usb_host_interface * alt alternate setting to search

struct usb_endpoint_descriptor ** bulk_in pointer to descriptor pointer, or NULL

struct usb_endpoint_descriptor ** bulk_out pointer to descriptor pointer, or NULL

struct usb_endpoint_descriptor ** int_in pointer to descriptor pointer, or NULL

struct usb_endpoint_descriptor ** int_out pointer to descriptor pointer, or NULL

Description

Search the alternate setting's endpoint descriptors for the last bulk-in, bulk-out, interrupt-in and interrupt-out endpoints and return them in the provided pointers (unless they are NULL).

If a requested endpoint is not found, the corresponding pointer is set to NULL.

Return

Zero if all requested descriptors were found, or -ENXIO otherwise.

struct usb_host_interface * **usb_find_alt_setting**(struct *usb_host_config* * *config*, unsigned int *iface_num*, unsigned int *alt_num*)

Given a configuration, find the alternate setting for the given interface.

Parameters

struct usb_host_config * config the configuration to search (not necessarily the current config).

unsigned int iface_num interface number to search in

unsigned int alt_num alternate interface setting number to search for.

Description

Search the configuration's interface cache for the given alt setting.

Return

The alternate setting, if found. NULL otherwise.

struct *usb_interface* * **usb_ifnum_to_if**(const struct *usb_device* * *dev*, unsigned *ifnum*)

get the interface object with a given interface number

Parameters

const struct usb_device * dev the device whose current configuration is considered

unsigned ifnum the desired interface

Description

This walks the device descriptor for the currently active configuration to find the interface object with the particular interface number.

Note that configuration descriptors are not required to assign interface numbers sequentially, so that it would be incorrect to assume that the first interface in that descriptor corresponds to interface zero. This routine helps device drivers avoid such mistakes. However, you should make sure that you do the right thing with any alternate settings available for this interfaces.

Don't call this function unless you are bound to one of the interfaces on this device or you have locked the device!

Return

A pointer to the interface that has **ifnum** as interface number, if found. `NULL` otherwise.

struct usb_host_interface * **`usb_altnum_to_altsetting`**(const struct *usb_interface* * *intf*, unsigned int *altnum*)
get the altsetting structure with a given alternate setting number.

Parameters

`const struct usb_interface * intf` the interface containing the altsetting in question

`unsigned int altnum` the desired alternate setting number

Description

This searches the altsetting array of the specified interface for an entry with the correct bAlternateSetting value.

Note that altsettings need not be stored sequentially by number, so it would be incorrect to assume that the first altsetting entry in the array corresponds to altsetting zero. This routine helps device drivers avoid such mistakes.

Don't call this function unless you are bound to the intf interface or you have locked the device!

Return

A pointer to the entry of the altsetting array of **intf** that has **altnum** as the alternate setting number. `NULL` if not found.

struct *usb_interface* * **`usb_find_interface`**(struct *usb_driver* * *drv*, int *minor*)
find usb_interface pointer for driver and device

Parameters

`struct usb_driver * drv` the driver whose current configuration is considered

`int minor` the minor number of the desired device

Description

This walks the bus device list and returns a pointer to the interface with the matching minor and driver. Note, this only works for devices that share the USB major number.

Return

A pointer to the interface with the matching major and **minor**.

int **`usb_for_each_dev`**(void * *data*, int (*fn) (struct *usb_device* *, void *)
iterate over all USB devices in the system

Parameters

`void * data` data pointer that will be handed to the callback function

`int (*)(struct usb_device *, void *) fn` callback function to be called for each USB device

Description

Iterate over all USB devices and call **fn** for each, passing it **data**. If it returns anything other than 0, we break the iteration prematurely and return that value.

struct *usb_device* * **usb_alloc_dev**(struct *usb_device* * *parent*, struct usb_bus * *bus*, unsigned *port1*)
usb device constructor (usbcore-internal)

Parameters

struct usb_device * parent hub to which device is connected; null to allocate a root hub

struct usb_bus * bus bus used to access the device

unsigned port1 one-based index of port; ignored for root hubs

Context

!:c:func:*in_interrupt()*

Description

Only hub drivers (including virtual root hub drivers for host controllers) should ever call this.

This call may not be used in a non-sleeping context.

Return

On success, a pointer to the allocated usb device. `NULL` on failure.

struct *usb_device* * **usb_get_dev**(struct *usb_device* * *dev*)
increments the reference count of the usb device structure

Parameters

struct usb_device * dev the device being referenced

Description

Each live reference to a device should be refcounted.

Drivers for USB interfaces should normally record such references in their `probe()` methods, when they bind to an interface, and release them by calling *`usb_put_dev()`*, in their `disconnect()` methods.

Return

A pointer to the device with the incremented reference counter.

void **usb_put_dev**(struct *usb_device* * *dev*)
release a use of the usb device structure

Parameters

struct usb_device * dev device that's been disconnected

Description

Must be called when a user of a device is finished with it. When the last user of the device calls this function, the memory of the device is freed.

struct *usb_interface* * **usb_get_intf**(struct *usb_interface* * *intf*)
increments the reference count of the usb interface structure

Parameters

struct usb_interface * intf the interface being referenced

Description

Each live reference to a interface must be refcounted.

Drivers for USB interfaces should normally record such references in their `probe()` methods, when they bind to an interface, and release them by calling *`usb_put_intf()`*, in their `disconnect()` methods.

Return

A pointer to the interface with the incremented reference counter.

void **`usb_put_intf`**(struct *usb_interface* * *intf*)
release a use of the usb interface structure

Parameters

`struct usb_interface * intf` interface that's been decremented

Description

Must be called when a user of an interface is finished with it. When the last user of the interface calls this function, the memory of the interface is freed.

int **`usb_lock_device_for_reset`**(struct *usb_device* * *udev*, const struct *usb_interface* * *iface*)
cautiously acquire the lock for a usb device structure

Parameters

`struct usb_device * udev` device that's being locked

`const struct usb_interface * iface` interface bound to the driver making the request (optional)

Description

Attempts to acquire the device lock, but fails if the device is NOTATTACHED or SUSPENDED, or if iface is specified and the interface is neither BINDING nor BOUND. Rather than sleeping to wait for the lock, the routine polls repeatedly. This is to prevent deadlock with disconnect; in some drivers (such as usb-storage) the `disconnect()` or `suspend()` method will block waiting for a device reset to complete.

Return

A negative error code for failure, otherwise 0.

int **`usb_get_current_frame_number`**(struct *usb_device* * *dev*)
return current bus frame number

Parameters

`struct usb_device * dev` the device whose bus is being queried

Return

The current frame number for the USB host controller used with the given USB device. This can be used when scheduling isochronous requests.

Note

Different kinds of host controller have different "scheduling horizons". While one type might support scheduling only 32 frames into the future, others could support scheduling up to 1024 frames into the future.

void * **`usb_alloc_coherent`**(struct *usb_device* * *dev*, size_t *size*, gfp_t *mem_flags*, dma_addr_t * *dma*)
allocate dma-consistent buffer for URB_NO_xxx_DMA_MAP

Parameters

`struct usb_device * dev` device the buffer will be used with

`size_t size` requested buffer size

`gfp_t mem_flags` affect whether allocation may block

`dma_addr_t * dma` used to return DMA address of buffer

Return

Either null (indicating no buffer could be allocated), or the cpu-space pointer to a buffer that may be used to perform DMA to the specified device. Such cpu-space buffers are returned along with the DMA address (through the pointer provided).

Note

These buffers are used with URB_NO_xxx_DMA_MAP set in urb->transfer_flags to avoid behaviors like using "DMA bounce buffers", or thrashing IOMMU hardware during URB completion/resubmit. The implementation varies between platforms, depending on details of how DMA will work to this device. Using these buffers also eliminates cacheline sharing problems on architectures where CPU caches are not DMA-coherent. On systems without bus-snooping caches, these buffers are uncached.

When the buffer is no longer used, free it with `usb_free_coherent()`.

void **usb_free_coherent**(struct usb_device * *dev*, size_t *size*, void * *addr*, dma_addr_t *dma*)
free memory allocated with `usb_alloc_coherent()`

Parameters

struct usb_device * dev device the buffer was used with

size_t size requested buffer size

void * addr CPU address of buffer

dma_addr_t dma DMA address of buffer

Description

This reclaims an I/O buffer, letting it be reused. The memory must have been allocated using `usb_alloc_coherent()`, and the parameters must match those provided in that allocation request.

struct urb * **usb_buffer_map**(struct urb * *urb*)
create DMA mapping(s) for an urb

Parameters

struct urb * urb urb whose transfer_buffer/setup_packet will be mapped

Description

URB_NO_TRANSFER_DMA_MAP is added to urb->transfer_flags if the operation succeeds. If the device is connected to this system through a non-DMA controller, this operation always succeeds.

This call would normally be used for an urb which is reused, perhaps as the target of a large periodic transfer, with `usb_buffer_dmasync()` calls to synchronize memory and dma state.

Reverse the effect of this call with `usb_buffer_unmap()`.

Return

Either `NULL` (indicating no buffer could be mapped), or **urb**.

void **usb_buffer_dmasync**(struct urb * *urb*)
synchronize DMA and CPU view of buffer(s)

Parameters

struct urb * urb urb whose transfer_buffer/setup_packet will be synchronized

void **usb_buffer_unmap**(struct urb * *urb*)
free DMA mapping(s) for an urb

Parameters

struct urb * urb urb whose transfer_buffer will be unmapped

Description

Reverses the effect of `usb_buffer_map()`.

int **usb_buffer_map_sg**(const struct usb_device * *dev*, int *is_in*, struct scatterlist * *sg*, int *nents*)
create scatterlist DMA mapping(s) for an endpoint

Parameters

const struct usb_device * dev device to which the scatterlist will be mapped

int is_in mapping transfer direction

struct scatterlist * sg the scatterlist to map

int nents the number of entries in the scatterlist

Return

Either < 0 (indicating no buffers could be mapped), or the number of DMA mapping array entries in the scatterlist.

Note

The caller is responsible for placing the resulting DMA addresses from the scatterlist into URB transfer buffer pointers, and for setting the URB_NO_TRANSFER_DMA_MAP transfer flag in each of those URBs.

Top I/O rates come from queuing URBs, instead of waiting for each one to complete before starting the next I/O. This is particularly easy to do with scatterlists. Just allocate and submit one URB for each DMA mapping entry returned, stopping on the first error or when all succeed. Better yet, use the usb_sg_*() calls, which do that (and more) for you.

This call would normally be used when translating scatterlist requests, rather than *usb_buffer_map()*, since on some hardware (with IOMMUs) it may be able to coalesce mappings for improved I/O efficiency.

Reverse the effect of this call with *usb_buffer_unmap_sg()*.

void **usb_buffer_dmasync_sg**(const struct *usb_device* * *dev*, int *is_in*, struct scatterlist * *sg*, int *n_hw_ents*)
synchronize DMA and CPU view of scatterlist buffer(s)

Parameters

const struct usb_device * dev device to which the scatterlist will be mapped

int is_in mapping transfer direction

struct scatterlist * sg the scatterlist to synchronize

int n_hw_ents the positive return value from usb_buffer_map_sg

Description

Use this when you are re-using a scatterlist's data buffers for another USB request.

void **usb_buffer_unmap_sg**(const struct *usb_device* * *dev*, int *is_in*, struct scatterlist * *sg*, int *n_hw_ents*)
free DMA mapping(s) for a scatterlist

Parameters

const struct usb_device * dev device to which the scatterlist will be mapped

int is_in mapping transfer direction

struct scatterlist * sg the scatterlist to unmap

int n_hw_ents the positive return value from usb_buffer_map_sg

Description

Reverses the effect of *usb_buffer_map_sg()*.

int **usb_hub_clear_tt_buffer**(struct *urb* * *urb*)
clear control/bulk TT state in high speed hub

Parameters

struct urb * urb an URB associated with the failed or incomplete split transaction

Description

High speed HCDs use this to tell the hub driver that some split control or bulk transaction failed in a way that requires clearing internal state of a transaction translator. This is normally detected (and reported) from interrupt context.

It may not be possible for that hub to handle additional full (or low) speed transactions until that state is fully cleared out.

Return

0 if successful. A negative error code otherwise.

void **usb_set_device_state**(struct *usb_device* * *udev*, enum usb_device_state *new_state*)
change a device's current state (usbcore, hcds)

Parameters

struct usb_device * udev pointer to device whose state should be changed

enum usb_device_state new_state new state value to be stored

Description

udev->state is _not_ fully protected by the device lock. Although most transitions are made only while holding the lock, the state can can change to USB_STATE_NOTATTACHED at almost any time. This is so that devices can be marked as disconnected as soon as possible, without having to wait for any semaphores to be released. As a result, all changes to any device's state must be protected by the device_state_lock spinlock.

Once a device has been added to the device tree, all changes to its state should be made using this routine. The state should _not_ be set directly.

If udev->state is already USB_STATE_NOTATTACHED then no change is made. Otherwise udev->state is set to new_state, and if new_state is USB_STATE_NOTATTACHED then all of udev's descendants' states are also set to USB_STATE_NOTATTACHED.

void **usb_root_hub_lost_power**(struct *usb_device* * *rhdev*)
called by HCD if the root hub lost Vbus power

Parameters

struct usb_device * rhdev struct usb_device for the root hub

Description

The USB host controller driver calls this function when its root hub is resumed and Vbus power has been interrupted or the controller has been reset. The routine marks **rhdev** as having lost power. When the hub driver is resumed it will take notice and carry out power-session recovery for all the "USB-PERSIST"-enabled child devices; the others will be disconnected.

int **usb_reset_device**(struct *usb_device* * *udev*)
warn interface drivers and perform a USB port reset

Parameters

struct usb_device * udev device to reset (not in SUSPENDED or NOTATTACHED state)

Description

Warns all drivers bound to registered interfaces (using their pre_reset method), performs the port reset, and then lets the drivers know that the reset is over (using their post_reset method).

Return

The same as for `usb_reset_and_verify_device()`.

Note

The caller must own the device lock. For example, it's safe to use this from a driver `probe()` routine after downloading new firmware. For calls that might not occur during `probe()`, drivers should lock the device using *`usb_lock_device_for_reset()`*.

If an interface is currently being probed or disconnected, we assume its driver knows how to handle resets. For all other interfaces, if the driver doesn't have pre_reset and post_reset methods then we attempt to unbind it and rebind afterward.

void **usb_queue_reset_device**(struct *usb_interface* * *iface*)
Reset a USB device from an atomic context

Parameters

struct usb_interface * iface USB interface belonging to the device to reset

Description

This function can be used to reset a USB device from an atomic context, where *usb_reset_device()* won't work (as it blocks).

Doing a reset via this method is functionally equivalent to calling *usb_reset_device()*, except for the fact that it is delayed to a workqueue. This means that any drivers bound to other interfaces might be unbound, as well as users from usbfs in user space.

Corner cases:

- Scheduling two resets at the same time from two different drivers attached to two different interfaces of the same device is possible; depending on how the driver attached to each interface handles ->:c:func:*pre_reset()*, the second reset might happen or not.
- If the reset is delayed so long that the interface is unbound from its driver, the reset will be skipped.
- This function can be called during .:c:func:*probe()*. It can also be called during .:c:func:*disconnect()*, but doing so is pointless because the reset will not occur. If you really want to reset the device during .:c:func:*disconnect()*, call *usb_reset_device()* directly – but watch out for nested unbinding issues!

struct *usb_device* * **usb_hub_find_child**(struct *usb_device* * *hdev*, int *port1*)
Get the pointer of child device attached to the port which is specified by **port1**.

Parameters

struct usb_device * hdev USB device belonging to the usb hub

int port1 port num to indicate which port the child device is attached to.

Description

USB drivers call this function to get hub's child device pointer.

Return

NULL if input param is invalid and child's usb_device pointer if non-NULL.

Host Controller APIs

These APIs are only for use by host controller drivers, most of which implement standard register interfaces such as XHCI, EHCI, OHCI, or UHCI. UHCI was one of the first interfaces, designed by Intel and also used by VIA; it doesn't do much in hardware. OHCI was designed later, to have the hardware do more work (bigger transfers, tracking protocol state, and so on). EHCI was designed with USB 2.0; its design has features that resemble OHCI (hardware does much more work) as well as UHCI (some parts of ISO support, TD list processing). XHCI was designed with USB 3.0. It continues to shift support for functionality into hardware.

There are host controllers other than the "big three", although most PCI based controllers (and a few non-PCI based ones) use one of those interfaces. Not all host controllers use DMA; some use PIO, and there is also a simulator and a virtual host controller to pipe USB over the network.

The same basic APIs are available to drivers for all those controllers. For historical reasons they are in two layers: `struct usb_bus` is a rather thin layer that became available in the 2.2 kernels, while `struct usb_hcd` is a more featureful layer that lets HCDs share common code, to shrink driver size and significantly reduce hcd-specific behaviors.

long **`usb_calc_bus_time`**(int *speed*, int *is_input*, int *isoc*, int *bytecount*)
approximate periodic transaction time in nanoseconds

Parameters

`int speed` from dev->speed; USB_SPEED_{LOW,FULL,HIGH}

`int is_input` true iff the transaction sends data to the host

`int isoc` true for isochronous transactions, false for interrupt ones

`int bytecount` how many bytes in the transaction.

Return

Approximate bus time in nanoseconds for a periodic transaction.

Note

See USB 2.0 spec section 5.11.3; only periodic transfers need to be scheduled in software, this function is only used for such scheduling.

int **`usb_hcd_link_urb_to_ep`**(struct usb_hcd * *hcd*, struct *urb* * *urb*)
add an URB to its endpoint queue

Parameters

`struct usb_hcd * hcd` host controller to which **urb** was submitted

`struct urb * urb` URB being submitted

Description

Host controller drivers should call this routine in their `enqueue()` method. The HCD's private spinlock must be held and interrupts must be disabled. The actions carried out here are required for URB submission, as well as for endpoint shutdown and for usb_kill_urb.

Return

0 for no error, otherwise a negative error code (in which case the `enqueue()` method must fail). If no error occurs but `enqueue()` fails anyway, it must call *`usb_hcd_unlink_urb_from_ep()`* before releasing the private spinlock and returning.

int **`usb_hcd_check_unlink_urb`**(struct usb_hcd * *hcd*, struct *urb* * *urb*, int *status*)
check whether an URB may be unlinked

Parameters

`struct usb_hcd * hcd` host controller to which **urb** was submitted

`struct urb * urb` URB being checked for unlinkability

`int status` error code to store in **urb** if the unlink succeeds

Description

Host controller drivers should call this routine in their `dequeue()` method. The HCD's private spinlock must be held and interrupts must be disabled. The actions carried out here are required for making sure than an unlink is valid.

Return

0 for no error, otherwise a negative error code (in which case the `dequeue()` method must fail). The possible error codes are:

-EIDRM: urb was not submitted or has already completed. The completion function may not have been called yet.

-EBUSY: **urb** has already been unlinked.

void **`usb_hcd_unlink_urb_from_ep`**(struct usb_hcd * *hcd*, struct *urb* * *urb*)
remove an URB from its endpoint queue

Parameters

struct usb_hcd * hcd host controller to which **urb** was submitted

struct urb * urb URB being unlinked

Description

Host controller drivers should call this routine before calling *usb_hcd_giveback_urb()*. The HCD's private spinlock must be held and interrupts must be disabled. The actions carried out here are required for URB completion.

void **usb_hcd_giveback_urb**(struct usb_hcd * *hcd*, struct urb * *urb*, int *status*)
return URB from HCD to device driver

Parameters

struct usb_hcd * hcd host controller returning the URB

struct urb * urb urb being returned to the USB device driver.

int status completion status code for the URB.

Context

`in_interrupt()`

Description

This hands the URB from HCD to its USB device driver, using its completion function. The HCD has freed all per-urb resources (and is done using urb->hcpriv). It also released all HCD locks; the device driver won't cause problems if it frees, modifies, or resubmits this URB.

If **urb** was unlinked, the value of **status** will be overridden by **urb**->unlinked. Erroneous short transfers are detected in case the HCD hasn't checked for them.

int **usb_alloc_streams**(struct usb_interface * *interface*, struct usb_host_endpoint ** *eps*, unsigned int *num_eps*, unsigned int *num_streams*, gfp_t *mem_flags*)
allocate bulk endpoint stream IDs.

Parameters

struct usb_interface * interface alternate setting that includes all endpoints.

struct usb_host_endpoint ** eps array of endpoints that need streams.

unsigned int num_eps number of endpoints in the array.

unsigned int num_streams number of streams to allocate.

gfp_t mem_flags flags hcd should use to allocate memory.

Description

Sets up a group of bulk endpoints to have **num_streams** stream IDs available. Drivers may queue multiple transfers to different stream IDs, which may complete in a different order than they were queued.

Return

On success, the number of allocated streams. On failure, a negative error code.

int **usb_free_streams**(struct usb_interface * *interface*, struct usb_host_endpoint ** *eps*, unsigned int *num_eps*, gfp_t *mem_flags*)
free bulk endpoint stream IDs.

Parameters

struct usb_interface * interface alternate setting that includes all endpoints.

struct usb_host_endpoint ** eps array of endpoints to remove streams from.

unsigned int num_eps number of endpoints in the array.

gfp_t mem_flags flags hcd should use to allocate memory.

Description

Reverts a group of bulk endpoints back to not using stream IDs. Can fail if we are given bad arguments, or HCD is broken.

Return

0 on success. On failure, a negative error code.

void **usb_hcd_resume_root_hub**(struct usb_hcd * *hcd*)
called by HCD to resume its root hub

Parameters

struct usb_hcd * hcd host controller for this root hub

Description

The USB host controller calls this function when its root hub is suspended (with the remote wakeup feature enabled) and a remote wakeup request is received. The routine submits a workqueue request to resume the root hub (that is, manage its downstream ports again).

int **usb_bus_start_enum**(struct usb_bus * *bus*, unsigned *port_num*)
start immediate enumeration (for OTG)

Parameters

struct usb_bus * bus the bus (must use hcd framework)

unsigned port_num 1-based number of port; usually bus->otg_port

Context

`in_interrupt()`

Description

Starts enumeration, with an immediate reset followed later by hub_wq identifying and possibly configuring the device. This is needed by OTG controller drivers, where it helps meet HNP protocol timing requirements for starting a port reset.

Return

0 if successful.

irqreturn_t **usb_hcd_irq**(int *irq*, void * *__hcd*)
hook IRQs to HCD framework (bus glue)

Parameters

int irq the IRQ being raised

void * __hcd pointer to the HCD whose IRQ is being signaled

Description

If the controller isn't HALTed, calls the driver's irq handler. Checks whether the controller is now dead.

Return

`IRQ_HANDLED` if the IRQ was handled. `IRQ_NONE` otherwise.

void **usb_hc_died**(struct usb_hcd * *hcd*)
report abnormal shutdown of a host controller (bus glue)

Parameters

struct usb_hcd * hcd pointer to the HCD representing the controller

Description

This is called by bus glue to report a USB host controller that died while operations may still have been pending. It's called automatically by the PCI glue, so only glue for non-PCI busses should need to call it.

Only call this function with the primary HCD.

struct usb_hcd * **`usb_create_shared_hcd`**(const struct hc_driver * *driver*, struct *device* * *dev*, const char * *bus_name*, struct usb_hcd * *primary_hcd*)
create and initialize an HCD structure

Parameters

`const struct hc_driver * driver` HC driver that will use this hcd

`struct device * dev` device for this HC, stored in hcd->self.controller

`const char * bus_name` value to store in hcd->self.bus_name

`struct usb_hcd * primary_hcd` a pointer to the usb_hcd structure that is sharing the PCI device. Only allocate certain resources for the primary HCD

Context

!:c:func:*in_interrupt()*

Description

Allocate a struct usb_hcd, with extra space at the end for the HC driver's private data. Initialize the generic members of the hcd structure.

Return

On success, a pointer to the created and initialized HCD structure. On failure (e.g. if memory is unavailable), `NULL`.

struct usb_hcd * **`usb_create_hcd`**(const struct hc_driver * *driver*, struct *device* * *dev*, const char * *bus_name*)
create and initialize an HCD structure

Parameters

`const struct hc_driver * driver` HC driver that will use this hcd

`struct device * dev` device for this HC, stored in hcd->self.controller

`const char * bus_name` value to store in hcd->self.bus_name

Context

!:c:func:*in_interrupt()*

Description

Allocate a struct usb_hcd, with extra space at the end for the HC driver's private data. Initialize the generic members of the hcd structure.

Return

On success, a pointer to the created and initialized HCD structure. On failure (e.g. if memory is unavailable), `NULL`.

int **`usb_add_hcd`**(struct usb_hcd * *hcd*, unsigned int *irqnum*, unsigned long *irqflags*)
finish generic HCD structure initialization and register

Parameters

`struct usb_hcd * hcd` the usb_hcd structure to initialize

`unsigned int irqnum` Interrupt line to allocate

`unsigned long irqflags` Interrupt type flags

Description

Finish the remaining parts of generic HCD initialization: allocate the buffers of consistent memory, register the bus, request the IRQ line, and call the driver's `reset()` and `start()` routines.

void **`usb_remove_hcd`**(struct usb_hcd * *hcd*)
shutdown processing for generic HCDs

Parameters

`struct usb_hcd * hcd` the usb_hcd structure to remove

Context

!:c:func:*in_interrupt()*

Description

Disconnects the root hub, then reverses the effects of *`usb_add_hcd()`*, invoking the HCD's `stop()` method.

int **`usb_hcd_pci_probe`**(struct pci_dev * *dev*, const struct pci_device_id * *id*)
initialize PCI-based HCDs

Parameters

`struct pci_dev * dev` USB Host Controller being probed

`const struct pci_device_id * id` pci hotplug id connecting controller to HCD framework

Context

!:c:func:*in_interrupt()*

Description

Allocates basic PCI resources for this USB host controller, and then invokes the `start()` method for the HCD associated with it through the hotplug entry's driver_data.

Store this function in the HCD's struct pci_driver as `probe()`.

Return

0 if successful.

void **`usb_hcd_pci_remove`**(struct pci_dev * *dev*)
shutdown processing for PCI-based HCDs

Parameters

`struct pci_dev * dev` USB Host Controller being removed

Context

!:c:func:*in_interrupt()*

Description

Reverses the effect of *`usb_hcd_pci_probe()`*, first invoking the HCD's `stop()` method. It is always called from a thread context, normally "rmmod", "apmd", or something similar.

Store this function in the HCD's struct pci_driver as `remove()`.

void **`usb_hcd_pci_shutdown`**(struct pci_dev * *dev*)
shutdown host controller

Parameters

`struct pci_dev * dev` USB Host Controller being shutdown

int **`hcd_buffer_create`**(struct usb_hcd * *hcd*)
initialize buffer pools

Parameters

`struct usb_hcd * hcd` the bus whose buffer pools are to be initialized

Context

!:c:func:*in_interrupt()*

Description

Call this as part of initializing a host controller that uses the dma memory allocators. It initializes some pools of dma-coherent memory that will be shared by all drivers using that controller.

Call `hcd_buffer_destroy()` to clean up after using those pools.

Return

0 if successful. A negative errno value otherwise.

void **hcd_buffer_destroy**(struct usb_hcd * *hcd*)
deallocate buffer pools

Parameters

struct usb_hcd * hcd the bus whose buffer pools are to be destroyed

Context

!:c:func:*in_interrupt()*

Description

This frees the buffer pools created by `hcd_buffer_create()`.

The USB character device nodes

This chapter presents the Linux character device nodes. You may prefer to avoid writing new kernel code for your USB driver. User mode device drivers are usually packaged as applications or libraries, and may use character devices through some programming library that wraps it. Such libraries include:

- libusb for C/C++, and
- jUSB for Java.

Some old information about it can be seen at the "USB Device Filesystem" section of the USB Guide. The latest copy of the USB Guide can be found at http://www.linux-usb.org/

Note:

- *They were used to be implemented via usbfs, but this is not part of the sysfs debug interface.*
- *This particular documentation is incomplete, especially with respect to the asynchronous mode. As of kernel 2.5.66 the code and this (new) documentation need to be cross-reviewed.*

What files are in "devtmpfs"?

Conventionally mounted at `/dev/bus/usb/`, usbfs features include:

- `/dev/bus/usb/BBB/DDD` … magic files exposing the each device's configuration descriptors, and supporting a series of ioctls for making device requests, including I/O to devices. (Purely for access by programs.)

Each bus is given a number (`BBB`) based on when it was enumerated; within each bus, each device is given a similar number (`DDD`). Those `BBB/DDD` paths are not "stable" identifiers; expect them to change even if you always leave the devices plugged in to the same hub port. *Don't even think of saving these in application configuration files.* Stable identifiers are available, for user mode applications that want to use them. HID and networking devices expose these stable IDs, so that for example you can be sure that you told the right UPS to power down its second server. Pleast note that it doesn't (yet) expose those IDs.

/dev/bus/usb/BBB/DDD

Use these files in one of these basic ways:

- *They can be read,* producing first the device descriptor (18 bytes) and then the descriptors for the current configuration. See the USB 2.0 spec for details about those binary data formats. You'll need to convert most multibyte values from little endian format to your native host byte order, although a few of the fields in the device descriptor (both of the BCD-encoded fields, and the vendor and product IDs) will be byteswapped for you. Note that configuration descriptors include descriptors for interfaces, altsettings, endpoints, and maybe additional class descriptors.
- *Perform USB operations* using *ioctl()* requests to make endpoint I/O requests (synchronously or asynchronously) or manage the device. These requests need the `CAP_SYS_RAWIO` capability, as well as filesystem access permissions. Only one ioctl request can be made on one of these device files at a time. This means that if you are synchronously reading an endpoint from one thread, you won't be able to write to a different endpoint from another thread until the read completes. This works for *half duplex* protocols, but otherwise you'd use asynchronous i/o requests.

Each connected USB device has one file. The `BBB` indicates the bus number. The `DDD` indicates the device address on that bus. Both of these numbers are assigned sequentially, and can be reused, so you can't rely on them for stable access to devices. For example, it's relatively common for devices to re-enumerate while they are still connected (perhaps someone jostled their power supply, hub, or USB cable), so a device might be `002/027` when you first connect it and `002/048` sometime later.

These files can be read as binary data. The binary data consists of first the device descriptor, then the descriptors for each configuration of the device. Multi-byte fields in the device descriptor are converted to host endianness by the kernel. The configuration descriptors are in bus endian format! The configuration descriptor are wTotalLength bytes apart. If a device returns less configuration descriptor data than indicated by wTotalLength there will be a hole in the file for the missing bytes. This information is also shown in text form by the `/sys/kernel/debug/usb/devices` file, described later.

These files may also be used to write user-level drivers for the USB devices. You would open the `/dev/bus/usb/BBB/DDD` file read/write, read its descriptors to make sure it's the device you expect, and then bind to an interface (or perhaps several) using an ioctl call. You would issue more ioctls to the device to communicate to it using control, bulk, or other kinds of USB transfers. The IOCTLs are listed in the `<linux/usbdevice_fs.h>` file, and at this writing the source code (`linux/drivers/usb/core/devio.c`) is the primary reference for how to access devices through those files.

Note that since by default these `BBB/DDD` files are writable only by root, only root can write such user mode drivers. You can selectively grant read/write permissions to other users by using `chmod`. Also, usbfs mount options such as `devmode=0666` may be helpful.

Life Cycle of User Mode Drivers

Such a driver first needs to find a device file for a device it knows how to handle. Maybe it was told about it because a `/sbin/hotplug` event handling agent chose that driver to handle the new device. Or maybe it's an application that scans all the `/dev/bus/usb` device files, and ignores most devices. In either case, it should `read()` all the descriptors from the device file, and check them against what it knows how to handle. It might just reject everything except a particular vendor and product ID, or need a more complex policy.

Never assume there will only be one such device on the system at a time! If your code can't handle more than one device at a time, at least detect when there's more than one, and have your users choose which device to use.

Once your user mode driver knows what device to use, it interacts with it in either of two styles. The simple style is to make only control requests; some devices don't need more complex interactions than those. (An example might be software using vendor-specific control requests for some initialization or configuration tasks, with a kernel driver for the rest.)

More likely, you need a more complex style driver: one using non-control endpoints, reading or writing data and claiming exclusive use of an interface. *Bulk* transfers are easiest to use, but only their sibling *interrupt* transfers work with low speed devices. Both interrupt and *isochronous* transfers offer service guarantees because their bandwidth is reserved. Such "periodic" transfers are awkward to use through usbfs, unless you're using the asynchronous calls. However, interrupt transfers can also be used in a synchronous "one shot" style.

Your user-mode driver should never need to worry about cleaning up request state when the device is disconnected, although it should close its open file descriptors as soon as it starts seeing the ENODEV errors.

The ioctl() Requests

To use these ioctls, you need to include the following headers in your userspace program:

```
#include <linux/usb.h>
#include <linux/usbdevice_fs.h>
#include <asm/byteorder.h>
```

The standard USB device model requests, from "Chapter 9" of the USB 2.0 specification, are automatically included from the `<linux/usb/ch9.h>` header.

Unless noted otherwise, the ioctl requests described here will update the modification time on the usbfs file to which they are applied (unless they fail). A return of zero indicates success; otherwise, a standard USB error code is returned (These are documented in *USB Error codes*).

Each of these files multiplexes access to several I/O streams, one per endpoint. Each device has one control endpoint (endpoint zero) which supports a limited RPC style RPC access. Devices are configured by hub_wq (in the kernel) setting a device-wide *configuration* that affects things like power consumption and basic functionality. The endpoints are part of USB *interfaces*, which may have *altsettings* affecting things like which endpoints are available. Many devices only have a single configuration and interface, so drivers for them will ignore configurations and altsettings.

Management/Status Requests

A number of usbfs requests don't deal very directly with device I/O. They mostly relate to device management and status. These are all synchronous requests.

USBDEVFS_CLAIMINTERFACE This is used to force usbfs to claim a specific interface, which has not previously been claimed by usbfs or any other kernel driver. The ioctl parameter is an integer holding the number of the interface (bInterfaceNumber from descriptor).

Note that if your driver doesn't claim an interface before trying to use one of its endpoints, and no other driver has bound to it, then the interface is automatically claimed by usbfs.

This claim will be released by a RELEASEINTERFACE ioctl, or by closing the file descriptor. File modification time is not updated by this request.

USBDEVFS_CONNECTINFO Says whether the device is lowspeed. The ioctl parameter points to a structure like this:

```
    struct usbdevfs_connectinfo {
            unsigned int   devnum;
            unsigned char  slow;
    };
```

File modification time is not updated by this request.

You can't tell whether a "not slow" device is connected at high speed (480 MBit/sec) or just full speed (12 MBit/sec). You should know the devnum value already, it's the DDD value of the device file name.

USBDEVFS_GETDRIVER Returns the name of the kernel driver bound to a given interface (a string). Parameter is a pointer to this structure, which is modified:

```
struct usbdevfs_getdriver {
        unsigned int  interface;
        char          driver[USBDEVFS_MAXDRIVERNAME + 1];
};
```

File modification time is not updated by this request.

USBDEVFS_IOCTL Passes a request from userspace through to a kernel driver that has an ioctl entry in the *struct usb_driver* it registered:

```
struct usbdevfs_ioctl {
        int     ifno;
        int     ioctl_code;
        void    *data;
};

/* user mode call looks like this.
 * 'request' becomes the driver->ioctl() 'code' parameter.
 * the size of 'param' is encoded in 'request', and that data
 * is copied to or from the driver->ioctl() 'buf' parameter.
 */
static int
usbdev_ioctl (int fd, int ifno, unsigned request, void *param)
{
        struct usbdevfs_ioctl   wrapper;

        wrapper.ifno = ifno;
        wrapper.ioctl_code = request;
        wrapper.data = param;

        return ioctl (fd, USBDEVFS_IOCTL, &wrapper);
}
```

File modification time is not updated by this request.

This request lets kernel drivers talk to user mode code through filesystem operations even when they don't create a character or block special device. It's also been used to do things like ask devices what device special file should be used. Two pre-defined ioctls are used to disconnect and reconnect kernel drivers, so that user mode code can completely manage binding and configuration of devices.

USBDEVFS_RELEASEINTERFACE This is used to release the claim usbfs made on interface, either implicitly or because of a USBDEVFS_CLAIMINTERFACE call, before the file descriptor is closed. The ioctl parameter is an integer holding the number of the interface (bInterfaceNumber from descriptor); File modification time is not updated by this request.

Warning: *No security check is made to ensure that the task which made the claim is the one whi releasing it. This means that user mode driver may interfere other ones.*

USBDEVFS_RESETEP Resets the data toggle value for an endpoint (bulk or interrupt) to DATA0. The ioctl parameter is an integer endpoint number (1 to 15, as identified in the endpoint descriptor), with USB_DIR_IN added if the device's endpoint sends data to the host.

Warning: *Avoid using this request. It should probably be removed. Using it typically means the de and driver will lose toggle synchronization. If you really lost synchronization, you likely ne completely handshake with the device, using a request like CLEAR_HALT or SET_INTERFA*

USBDEVFS_DROP_PRIVILEGES This is used to relinquish the ability to do certain operations which are considered to be privileged on a usbfs file descriptor. This includes claiming arbitrary interfaces, resetting a device on which there are currently claimed interfaces from other users, and issuing

USBDEVFS_IOCTL calls. The ioctl parameter is a 32 bit mask of interfaces the user is allowed to claim on this file descriptor. You may issue this ioctl more than one time to narrow said mask.

Synchronous I/O Support

Synchronous requests involve the kernel blocking until the user mode request completes, either by finishing successfully or by reporting an error. In most cases this is the simplest way to use usbfs, although as noted above it does prevent performing I/O to more than one endpoint at a time.

USBDEVFS_BULK Issues a bulk read or write request to the device. The ioctl parameter is a pointer to this structure:

```
struct usbdevfs_bulktransfer {
        unsigned int  ep;
        unsigned int  len;
        unsigned int  timeout; /* in milliseconds */
        void          *data;
};
```

The ep value identifies a bulk endpoint number (1 to 15, as identified in an endpoint descriptor), masked with USB_DIR_IN when referring to an endpoint which sends data to the host from the device. The length of the data buffer is identified by `len`; Recent kernels support requests up to about 128KBytes. *FIXME say how read length is returned, and how short reads are handled.*.

USBDEVFS_CLEAR_HALT Clears endpoint halt (stall) and resets the endpoint toggle. This is only meaningful for bulk or interrupt endpoints. The ioctl parameter is an integer endpoint number (1 to 15, as identified in an endpoint descriptor), masked with USB_DIR_IN when referring to an endpoint which sends data to the host from the device.

Use this on bulk or interrupt endpoints which have stalled, returning -EPIPE status to a data transfer request. Do not issue the control request directly, since that could invalidate the host's record of the data toggle.

USBDEVFS_CONTROL Issues a control request to the device. The ioctl parameter points to a structure like this:

```
struct usbdevfs_ctrltransfer {
        __u8   bRequestType;
        __u8   bRequest;
        __u16  wValue;
        __u16  wIndex;
        __u16  wLength;
        __u32  timeout;  /* in milliseconds */
        void   *data;
};
```

The first eight bytes of this structure are the contents of the SETUP packet to be sent to the device; see the USB 2.0 specification for details. The bRequestType value is composed by combining a `USB_TYPE_*` value, a `USB_DIR_*` value, and a `USB_RECIP_*` value (from `linux/usb.h`). If wLength is nonzero, it describes the length of the data buffer, which is either written to the device (USB_DIR_OUT) or read from the device (USB_DIR_IN).

At this writing, you can't transfer more than 4 KBytes of data to or from a device; usbfs has a limit, and some host controller drivers have a limit. (That's not usually a problem.) *Also* there's no way to say it's not OK to get a short read back from the device.

USBDEVFS_RESET Does a USB level device reset. The ioctl parameter is ignored. After the reset, this rebinds all device interfaces. File modification time is not updated by this request.

Warning: *Avoid using this call until some usbcore bugs get fixed, since it does not fully synchronize de interface, and driver (not just usbfs) state.*

USBDEVFS_SETINTERFACE Sets the alternate setting for an interface. The ioctl parameter is a pointer to a structure like this:

```
struct usbdevfs_setinterface {
        unsigned int  interface;
        unsigned int  altsetting;
};
```

File modification time is not updated by this request.

Those struct members are from some interface descriptor applying to the current configuration. The interface number is the bInterfaceNumber value, and the altsetting number is the bAlternateSetting value. (This resets each endpoint in the interface.)

USBDEVFS_SETCONFIGURATION Issues the `usb_set_configuration()` call for the device. The parameter is an integer holding the number of a configuration (bConfigurationValue from descriptor). File modification time is not updated by this request.

Warning: *Avoid using this call until some usbcore bugs get fixed, since it does not fully synchronize de interface, and driver (not just usbfs) state.*

Asynchronous I/O Support

As mentioned above, there are situations where it may be important to initiate concurrent operations from user mode code. This is particularly important for periodic transfers (interrupt and isochronous), but it can be used for other kinds of USB requests too. In such cases, the asynchronous requests described here are essential. Rather than submitting one request and having the kernel block until it completes, the blocking is separate.

These requests are packaged into a structure that resembles the URB used by kernel device drivers. (No POSIX Async I/O support here, sorry.) It identifies the endpoint type (USBDEVFS_URB_TYPE_*), endpoint (number, masked with USB_DIR_IN as appropriate), buffer and length, and a user "context" value serving to uniquely identify each request. (It's usually a pointer to per-request data.) Flags can modify requests (not as many as supported for kernel drivers).

Each request can specify a realtime signal number (between SIGRTMIN and SIGRTMAX, inclusive) to request a signal be sent when the request completes.

When usbfs returns these urbs, the status value is updated, and the buffer may have been modified. Except for isochronous transfers, the actual_length is updated to say how many bytes were transferred; if the USBDEVFS_URB_DISABLE_SPD flag is set ("short packets are not OK"), if fewer bytes were read than were requested then you get an error report:

```
struct usbdevfs_iso_packet_desc {
        unsigned int                    length;
        unsigned int                    actual_length;
        unsigned int                    status;
};

struct usbdevfs_urb {
        unsigned char                   type;
        unsigned char                   endpoint;
        int                             status;
        unsigned int                    flags;
        void                            *buffer;
        int                             buffer_length;
        int                             actual_length;
        int                             start_frame;
        int                             number_of_packets;
        int                             error_count;
        unsigned int                    signr;
        void                            *usercontext;
        struct usbdevfs_iso_packet_desc  iso_frame_desc[];
};
```

For these asynchronous requests, the file modification time reflects when the request was initiated. This contrasts with their use with the synchronous requests, where it reflects when requests complete.

USBDEVFS_DISCARDURB *TBS* File modification time is not updated by this request.

USBDEVFS_DISCSIGNAL *TBS* File modification time is not updated by this request.

USBDEVFS_REAPURB *TBS* File modification time is not updated by this request.

USBDEVFS_REAPURBNDELAY *TBS* File modification time is not updated by this request.

USBDEVFS_SUBMITURB *TBS*

The USB devices

The USB devices are now exported via debugfs:

- `/sys/kernel/debug/usb/devices` … a text file showing each of the USB devices on known to the kernel, and their configuration descriptors. You can also poll() this to learn about new devices.

/sys/kernel/debug/usb/devices

This file is handy for status viewing tools in user mode, which can scan the text format and ignore most of it. More detailed device status (including class and vendor status) is available from device-specific files. For information about the current format of this file, see below.

This file, in combination with the poll() system call, can also be used to detect when devices are added or removed:

```
int fd;
struct pollfd pfd;

fd = open("/sys/kernel/debug/usb/devices", O_RDONLY);
pfd = { fd, POLLIN, 0 };
for (;;) {
    /* The first time through, this call will return immediately. */
    poll(&pfd, 1, -1);

    /* To see what's changed, compare the file's previous and current
       contents or scan the filesystem.  (Scanning is more precise.) */
}
```

Note that this behavior is intended to be used for informational and debug purposes. It would be more appropriate to use programs such as udev or HAL to initialize a device or start a user-mode helper program, for instance.

In this file, each device's output has multiple lines of ASCII output.

I made it ASCII instead of binary on purpose, so that someone can obtain some useful data from it without the use of an auxiliary program. However, with an auxiliary program, the numbers in the first 4 columns of each T: line (topology info: Lev, Prnt, Port, Cnt) can be used to build a USB topology diagram.

Each line is tagged with a one-character ID for that line:

```
T = Topology (etc.)
B = Bandwidth (applies only to USB host controllers, which are
virtualized as root hubs)
D = Device descriptor info.
```

```
P = Product ID info. (from Device descriptor, but they won't fit
together on one line)
S = String descriptors.
C = Configuration descriptor info. (* = active configuration)
I = Interface descriptor info.
E = Endpoint descriptor info.
```

/sys/kernel/debug/usb/devices output format

Legend:: d = decimal number (may have leading spaces or 0's) x = hexadecimal number (may have leading spaces or 0's) s = string

Topology info

```
T:  Bus=dd Lev=dd Prnt=dd Port=dd Cnt=dd Dev#=ddd Spd=dddd MxCh=dd
|   |      |      |       |       |      |        |         |__MaxChildren
|   |      |      |       |       |      |        |__Device Speed in Mbps
|   |      |      |       |       |      |__DeviceNumber
|   |      |      |       |       |__Count of devices at this level
|   |      |      |       |__Connector/Port on Parent for this device
|   |      |      |__Parent DeviceNumber
|   |      |__Level in topology for this bus
|   |__Bus number
|__Topology info tag
```

Speed may be:

1.5	Mbit/s for low speed USB
12	Mbit/s for full speed USB
480	Mbit/s for high speed USB (added for USB 2.0); also used for Wireless USB, which has no fixed speed
5000	Mbit/s for SuperSpeed USB (added for USB 3.0)

For reasons lost in the mists of time, the Port number is always too low by 1. For example, a device plugged into port 4 will show up with `Port=03`.

Bandwidth info

```
B:  Alloc=ddd/ddd us (xx%), #Int=ddd, #Iso=ddd
|   |                       |         |__Number of isochronous requests
|   |                       |__Number of interrupt requests
|   |__Total Bandwidth allocated to this bus
|__Bandwidth info tag
```

Bandwidth allocation is an approximation of how much of one frame (millisecond) is in use. It reflects only periodic transfers, which are the only transfers that reserve bandwidth. Control and bulk transfers use all other bandwidth, including reserved bandwidth that is not used for transfers (such as for short packets).

The percentage is how much of the "reserved" bandwidth is scheduled by those transfers. For a low or full speed bus (loosely, "USB 1.1"), 90% of the bus bandwidth is reserved. For a high speed bus (loosely, "USB 2.0") 80% is reserved.

Device descriptor info & Product ID info

```
D:  Ver=x.xx Cls=xx(s) Sub=xx Prot=xx MxPS=dd #Cfgs=dd
P:  Vendor=xxxx ProdID=xxxx Rev=xx.xx
```

where:

```
D:  Ver=x.xx Cls=xx(sssss) Sub=xx Prot=xx MxPS=dd #Cfgs=dd
|   |        |             |      |       |       |__NumberConfigurations
|   |        |             |      |       |__MaxPacketSize of Default Endpoint
|   |        |             |      |__DeviceProtocol
|   |        |             |__DeviceSubClass
|   |        |__DeviceClass
|   |__Device USB version
|__Device info tag #1
```

where:

```
P:  Vendor=xxxx ProdID=xxxx Rev=xx.xx
|   |           |           |__Product revision number
|   |           |__Product ID code
|   |__Vendor ID code
|__Device info tag #2
```

String descriptor info

```
S:  Manufacturer=ssss
|   |__Manufacturer of this device as read from the device.
|      For USB host controller drivers (virtual root hubs) this may
|      be omitted, or (for newer drivers) will identify the kernel
|      version and the driver which provides this hub emulation.
|__String info tag

S:  Product=ssss
|   |__Product description of this device as read from the device.
|      For older USB host controller drivers (virtual root hubs) this
|      indicates the driver; for newer ones, it's a product (and vendor)
|      description that often comes from the kernel's PCI ID database.
|__String info tag

S:  SerialNumber=ssss
|   |__Serial Number of this device as read from the device.
|      For USB host controller drivers (virtual root hubs) this is
|      some unique ID, normally a bus ID (address or slot name) that
|      can't be shared with any other device.
|__String info tag
```

Configuration descriptor info

```
C:* #Ifs=dd Cfg#=dd Atr=xx MPwr=dddmA
| | |       |       |      |__MaxPower in mA
| | |       |       |__Attributes
| | |       |__ConfiguratioNumber
| | |__NumberOfInterfaces
| |__ "*" indicates the active configuration (others are " ")
|__Config info tag
```

USB devices may have multiple configurations, each of which act rather differently. For example, a bus-powered configuration might be much less capable than one that is self-powered. Only one device configuration can be active at a time; most devices have only one configuration.

Each configuration consists of one or more interfaces. Each interface serves a distinct "function", which is typically bound to a different USB device driver. One common example is a USB speaker with an audio interface for playback, and a HID interface for use with software volume control.

Interface descriptor info (can be multiple per Config)

```
I:* If#=dd Alt=dd #EPs=dd Cls=xx(sssss) Sub=xx Prot=xx Driver=ssss
| | |      |      |       |             |      |       |__Driver name
| | |      |      |       |             |      |          or "(none)"
| | |      |      |       |             |      |__InterfaceProtocol
| | |      |      |       |             |__InterfaceSubClass
| | |      |      |       |__InterfaceClass
| | |      |      |__NumberOfEndpoints
| | |      |__AlternateSettingNumber
| | |__InterfaceNumber
| |__ "*" indicates the active altsetting (others are " ")
|__Interface info tag
```

A given interface may have one or more “alternate” settings. For example, default settings may not use more than a small amount of periodic bandwidth. To use significant fractions of bus bandwidth, drivers must select a non-default altsetting.

Only one setting for an interface may be active at a time, and only one driver may bind to an interface at a time. Most devices have only one alternate setting per interface.

Endpoint descriptor info (can be multiple per Interface)

```
E:  Ad=xx(s) Atr=xx(ssss) MxPS=dddd Ivl=dddss
|   |        |            |         |__Interval (max) between transfers
|   |        |            |__EndpointMaxPacketSize
|   |        |__Attributes(EndpointType)
|   |__EndpointAddress(I=In,O=Out)
|__Endpoint info tag
```

The interval is nonzero for all periodic (interrupt or isochronous) endpoints. For high speed endpoints the transfer interval may be measured in microseconds rather than milliseconds.

For high speed periodic endpoints, the `EndpointMaxPacketSize` reflects the per-microframe data transfer size. For “high bandwidth” endpoints, that can reflect two or three packets (for up to 3KBytes every 125 usec) per endpoint.

With the Linux-USB stack, periodic bandwidth reservations use the transfer intervals and sizes provided by URBs, which can be less than those found in endpoint descriptor.

Usage examples

If a user or script is interested only in Topology info, for example, use something like `grep ^T: /sys/kernel/debug/usb/devices` for only the Topology lines. A command like `grep -i ^[tdp]: /sys/kernel/debug/usb/devices` can be used to list only the lines that begin with the characters in square brackets, where the valid characters are TDPCIE. With a slightly more able script, it can display any selected lines (for example, only T, D, and P lines) and change their output format. (The `procusb` Perl script is the beginning of this idea. It will list only selected lines [selected from TBDPSCIE] or “All” lines from `/sys/kernel/debug/usb/devices`.)

The Topology lines can be used to generate a graphic/pictorial of the USB devices on a system’s root hub. (See more below on how to do this.)

The Interface lines can be used to determine what driver is being used for each device, and which altsetting it activated.

The Configuration lines could be used to list maximum power (in milliamps) that a system’s USB devices are using. For example, `grep ^C: /sys/kernel/debug/usb/devices`.

Here’s an example, from a system which has a UHCI root hub, an external hub connected to the root hub, and a mouse and a serial converter connected to the external hub.

```
T:  Bus=00 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#=  1 Spd=12   MxCh= 2
B:  Alloc= 28/900 us ( 3%), #Int=  2, #Iso=  0
```

```
D:  Ver= 1.00 Cls=09(hub  ) Sub=00 Prot=00 MxPS= 8 #Cfgs=  1
P:  Vendor=0000 ProdID=0000 Rev= 0.00
S:  Product=USB UHCI Root Hub
S:  SerialNumber=dce0
C:* #Ifs= 1 Cfg#= 1 Atr=40 MxPwr=  0mA
I:  If#= 0 Alt= 0 #EPs= 1 Cls=09(hub  ) Sub=00 Prot=00 Driver=hub
E:  Ad=81(I) Atr=03(Int.) MxPS=   8 Ivl=255ms

T:  Bus=00 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#=  2 Spd=12   MxCh= 4
D:  Ver= 1.00 Cls=09(hub  ) Sub=00 Prot=00 MxPS= 8 #Cfgs=  1
P:  Vendor=0451 ProdID=1446 Rev= 1.00
C:* #Ifs= 1 Cfg#= 1 Atr=e0 MxPwr=100mA
I:  If#= 0 Alt= 0 #EPs= 1 Cls=09(hub  ) Sub=00 Prot=00 Driver=hub
E:  Ad=81(I) Atr=03(Int.) MxPS=   1 Ivl=255ms

T:  Bus=00 Lev=02 Prnt=02 Port=00 Cnt=01 Dev#=  3 Spd=1.5  MxCh= 0
D:  Ver= 1.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS= 8 #Cfgs=  1
P:  Vendor=04b4 ProdID=0001 Rev= 0.00
C:* #Ifs= 1 Cfg#= 1 Atr=80 MxPwr=100mA
I:  If#= 0 Alt= 0 #EPs= 1 Cls=03(HID  ) Sub=01 Prot=02 Driver=mouse
E:  Ad=81(I) Atr=03(Int.) MxPS=   3 Ivl= 10ms

T:  Bus=00 Lev=02 Prnt=02 Port=02 Cnt=02 Dev#=  4 Spd=12   MxCh= 0
D:  Ver= 1.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS= 8 #Cfgs=  1
P:  Vendor=0565 ProdID=0001 Rev= 1.08
S:  Manufacturer=Peracom Networks, Inc.
S:  Product=Peracom USB to Serial Converter
C:* #Ifs= 1 Cfg#= 1 Atr=a0 MxPwr=100mA
I:  If#= 0 Alt= 0 #EPs= 3 Cls=00(>ifc ) Sub=00 Prot=00 Driver=serial
E:  Ad=81(I) Atr=02(Bulk) MxPS=  64 Ivl= 16ms
E:  Ad=01(O) Atr=02(Bulk) MxPS=  16 Ivl= 16ms
E:  Ad=82(I) Atr=03(Int.) MxPS=   8 Ivl=  8ms
```

Selecting only the T: and I: lines from this (for example, by using procusb ti), we have

```
T:  Bus=00 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#=  1 Spd=12   MxCh= 2
T:  Bus=00 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#=  2 Spd=12   MxCh= 4
I:  If#= 0 Alt= 0 #EPs= 1 Cls=09(hub  ) Sub=00 Prot=00 Driver=hub
T:  Bus=00 Lev=02 Prnt=02 Port=00 Cnt=01 Dev#=  3 Spd=1.5  MxCh= 0
I:  If#= 0 Alt= 0 #EPs= 1 Cls=03(HID  ) Sub=01 Prot=02 Driver=mouse
T:  Bus=00 Lev=02 Prnt=02 Port=02 Cnt=02 Dev#=  4 Spd=12   MxCh= 0
I:  If#= 0 Alt= 0 #EPs= 3 Cls=00(>ifc ) Sub=00 Prot=00 Driver=serial
```

Physically this looks like (or could be converted to):

```
                    +------------------+
                    |  PC/root_hub (12)|   Dev# = 1
                    +------------------+   (nn) is Mbps.
  Level 0           |  CN.0   |  CN.1  |   [CN = connector/port #]
                    +------------------+
                        /
                       /
          +-----------------------+
Level 1   | Dev#2: 4-port hub (12)|
          +-----------------------+
          |CN.0 |CN.1 |CN.2 |CN.3 |
          +-----------------------+
              \          \____________________
               \_____                         \
                     \                         \
             +--------------------+      +--------------------+
Level 2      | Dev# 3: mouse (1.5)|      | Dev# 4: serial (12)|
             +--------------------+      +--------------------+
```

Or, in a more tree-like structure (ports [Connectors] without connections could be omitted):

```
PC:  Dev# 1, root hub, 2 ports, 12 Mbps
|_ CN.0:  Dev# 2, hub, 4 ports, 12 Mbps
     |_ CN.0:  Dev #3, mouse, 1.5 Mbps
     |_ CN.1:
     |_ CN.2:  Dev #4, serial, 12 Mbps
     |_ CN.3:
|_ CN.1:
```

USB Gadget API for Linux

Author David Brownell

Date 20 August 2004

Introduction

This document presents a Linux-USB "Gadget" kernel mode API, for use within peripherals and other USB devices that embed Linux. It provides an overview of the API structure, and shows how that fits into a system development project. This is the first such API released on Linux to address a number of important problems, including:

- Supports USB 2.0, for high speed devices which can stream data at several dozen megabytes per second.
- Handles devices with dozens of endpoints just as well as ones with just two fixed-function ones. Gadget drivers can be written so they're easy to port to new hardware.
- Flexible enough to expose more complex USB device capabilities such as multiple configurations, multiple interfaces, composite devices, and alternate interface settings.
- USB "On-The-Go" (OTG) support, in conjunction with updates to the Linux-USB host side.
- Sharing data structures and API models with the Linux-USB host side API. This helps the OTG support, and looks forward to more-symmetric frameworks (where the same I/O model is used by both host and device side drivers).
- Minimalist, so it's easier to support new device controller hardware. I/O processing doesn't imply large demands for memory or CPU resources.

Most Linux developers will not be able to use this API, since they have USB `host` hardware in a PC, workstation, or server. Linux users with embedded systems are more likely to have USB peripheral hardware. To distinguish drivers running inside such hardware from the more familiar Linux "USB device drivers", which are host side proxies for the real USB devices, a different term is used: the drivers inside the peripherals are "USB gadget drivers". In USB protocol interactions, the device driver is the master (or "client driver") and the gadget driver is the slave (or "function driver").

The gadget API resembles the host side Linux-USB API in that both use queues of request objects to package I/O buffers, and those requests may be submitted or canceled. They share common definitions for the standard USB *Chapter 9* messages, structures, and constants. Also, both APIs bind and unbind drivers to devices. The APIs differ in detail, since the host side's current URB framework exposes a number of implementation details and assumptions that are inappropriate for a gadget API. While the model for control transfers and configuration management is necessarily different (one side is a hardware-neutral master, the other is a hardware-aware slave), the endpoint I/0 API used here should also be usable for an overhead-reduced host side API.

Structure of Gadget Drivers

A system running inside a USB peripheral normally has at least three layers inside the kernel to handle USB protocol processing, and may have additional layers in user space code. The `gadget` API is used by the middle layer to interact with the lowest level (which directly handles hardware).

In Linux, from the bottom up, these layers are:

USB Controller Driver This is the lowest software level. It is the only layer that talks to hardware, through registers, fifos, dma, irqs, and the like. The `<linux/usb/gadget.h>` API abstracts the peripheral controller endpoint hardware. That hardware is exposed through endpoint objects, which accept streams of IN/OUT buffers, and through callbacks that interact with gadget drivers. Since normal USB devices only have one upstream port, they only have one of these drivers. The controller driver can support any number of different gadget drivers, but only one of them can be used at a time.

Examples of such controller hardware include the PCI-based NetChip 2280 USB 2.0 high speed controller, the SA-11x0 or PXA-25x UDC (found within many PDAs), and a variety of other products.

Gadget Driver The lower boundary of this driver implements hardware-neutral USB functions, using calls to the controller driver. Because such hardware varies widely in capabilities and restrictions, and is used in embedded environments where space is at a premium, the gadget driver is often configured at compile time to work with endpoints supported by one particular controller. Gadget drivers may be portable to several different controllers, using conditional compilation. (Recent kernels substantially simplify the work involved in supporting new hardware, by *autoconfiguring* endpoints automatically for many bulk-oriented drivers.) Gadget driver responsibilities include:

- handling setup requests (ep0 protocol responses) possibly including class-specific functionality
- returning configuration and string descriptors
- (re)setting configurations and interface altsettings, including enabling and configuring endpoints
- handling life cycle events, such as managing bindings to hardware, USB suspend/resume, remote wakeup, and disconnection from the USB host.
- managing IN and OUT transfers on all currently enabled endpoints

Such drivers may be modules of proprietary code, although that approach is discouraged in the Linux community.

Upper Level Most gadget drivers have an upper boundary that connects to some Linux driver or framework in Linux. Through that boundary flows the data which the gadget driver produces and/or consumes through protocol transfers over USB. Examples include:

- user mode code, using generic (gadgetfs) or application specific files in `/dev`
- networking subsystem (for network gadgets, like the CDC Ethernet Model gadget driver)
- data capture drivers, perhaps video4Linux or a scanner driver; or test and measurement hardware.
- input subsystem (for HID gadgets)
- sound subsystem (for audio gadgets)
- file system (for PTP gadgets)
- block i/o subsystem (for usb-storage gadgets)
- ... and more

Additional Layers Other layers may exist. These could include kernel layers, such as network protocol stacks, as well as user mode applications building on standard POSIX system call APIs such as `open()`, `close()`, `read()` and `write()`. On newer systems, POSIX Async I/O calls may be an option. Such user mode code will not necessarily be subject to the GNU General Public License (GPL).

OTG-capable systems will also need to include a standard Linux-USB host side stack, with `usbcore`, one or more *Host Controller Drivers* (HCDs), *USB Device Drivers* to support the OTG "Targeted Peripheral List", and so forth. There will also be an *OTG Controller Driver*, which is visible to gadget and device driver developers only indirectly. That helps the host and device side USB controllers implement the two new OTG protocols (HNP and SRP). Roles switch (host to peripheral, or vice versa) using HNP during USB suspend processing, and SRP can be viewed as a more battery-friendly kind of device wakeup protocol.

Over time, reusable utilities are evolving to help make some gadget driver tasks simpler. For example, building configuration descriptors from vectors of descriptors for the configurations interfaces and endpoints is now automated, and many drivers now use autoconfiguration to choose hardware endpoints and initialize their descriptors. A potential example of particular interest is code implementing standard USB-IF protocols for HID, networking, storage, or audio classes. Some developers are interested in KDB or KGDB hooks, to let target hardware be remotely debugged. Most such USB protocol code doesn't need to be hardware-specific, any more than network protocols like X11, HTTP, or NFS are. Such gadget-side interface drivers should eventually be combined, to implement composite devices.

Kernel Mode Gadget API

Gadget drivers declare themselves through a struct *`usb_gadget_driver`*, which is responsible for most parts of enumeration for a struct *`usb_gadget`*. The response to a set_configuration usually involves enabling one or more of the struct *`usb_ep`* objects exposed by the gadget, and submitting one or more struct *`usb_request`* buffers to transfer data. Understand those four data types, and their operations, and you will understand how this API works.

Note:

Other than the "Chapter 9" data types, most of the significant data types and functions are described here.
However, some relevant information is likely omitted from what you are reading. One example of such information is endpoint autoconfiguration. You'll have to read the header file, and use example source code (such as that for "Gadget Zero"), to fully understand the API.
The part of the API implementing some basic driver capabilities is specific to the version of the Linux kernel that's in use. The 2.6 and upper kernel versions include a driver model framework that has no analogue on earlier kernels; so those parts of the gadget API are not fully portable. (They are implemented on 2.4 kernels, but in a different way.) The driver model state is another part of this API that is ignored by the kerneldoc tools.

The core API does not expose every possible hardware feature, only the most widely available ones. There are significant hardware features, such as device-to-device DMA (without temporary storage in a memory buffer) that would be added using hardware-specific APIs.

This API allows drivers to use conditional compilation to handle endpoint capabilities of different hardware, but doesn't require that. Hardware tends to have arbitrary restrictions, relating to transfer types, addressing, packet sizes, buffering, and availability. As a rule, such differences only matter for "endpoint zero" logic that handles device configuration and management. The API supports limited run-time detection of capabilities, through naming conventions for endpoints. Many drivers will be able to at least partially autoconfigure themselves. In particular, driver init sections will often have endpoint autoconfiguration logic that scans the hardware's list of endpoints to find ones matching the driver requirements (relying on those conventions), to eliminate some of the most common reasons for conditional compilation.

Like the Linux-USB host side API, this API exposes the "chunky" nature of USB messages: I/O requests are in terms of one or more "packets", and packet boundaries are visible to drivers. Compared to RS-232 serial protocols, USB resembles synchronous protocols like HDLC (N bytes per frame, multipoint addressing, host as the primary station and devices as secondary stations) more than asynchronous ones (tty style: 8 data bits per frame, no parity, one stop bit). So for example the controller drivers won't buffer two single byte writes into a single two-byte USB IN packet, although gadget drivers may do so when they implement protocols where packet boundaries (and "short packets") are not significant.

Driver Life Cycle

Gadget drivers make endpoint I/O requests to hardware without needing to know many details of the hardware, but driver setup/configuration code needs to handle some differences. Use the API like this:

1. Register a driver for the particular device side usb controller hardware, such as the net2280 on PCI (USB 2.0), sa11x0 or pxa25x as found in Linux PDAs, and so on. At this point the device is logically in the USB ch9 initial state (`attached`), drawing no power and not usable (since it does not yet support enumeration). Any host should not see the device, since it's not activated the data line pullup used by the host to detect a device, even if VBUS power is available.
2. Register a gadget driver that implements some higher level device function. That will then bind() to a *`usb_gadget`*, which activates the data line pullup sometime after detecting VBUS.
3. The hardware driver can now start enumerating. The steps it handles are to accept USB `power` and `set_address` requests. Other steps are handled by the gadget driver. If the gadget driver module is unloaded before the host starts to enumerate, steps before step 7 are skipped.
4. The gadget driver's `setup()` call returns usb descriptors, based both on what the bus interface hardware provides and on the functionality being implemented. That can involve alternate settings or configurations, unless the hardware prevents such operation. For OTG devices, each configuration descriptor includes an OTG descriptor.
5. The gadget driver handles the last step of enumeration, when the USB host issues a `set_configuration` call. It enables all endpoints used in that configuration, with all interfaces in their default settings. That involves using a list of the hardware's endpoints, enabling each endpoint according to its descriptor. It may also involve using `usb_gadget_vbus_draw` to let more power be drawn from VBUS, as allowed by that configuration. For OTG devices, setting a configuration may also involve reporting HNP capabilities through a user interface.
6. Do real work and perform data transfers, possibly involving changes to interface settings or switching to new configurations, until the device is disconnect()ed from the host. Queue any number of transfer requests to each endpoint. It may be suspended and resumed several times before being disconnected. On disconnect, the drivers go back to step 3 (above).
7. When the gadget driver module is being unloaded, the driver unbind() callback is issued. That lets the controller driver be unloaded.

Drivers will normally be arranged so that just loading the gadget driver module (or statically linking it into a Linux kernel) allows the peripheral device to be enumerated, but some drivers will defer enumeration until some higher level component (like a user mode daemon) enables it. Note that at this lowest level there are no policies about how ep0 configuration logic is implemented, except that it should obey USB specifications. Such issues are in the domain of gadget drivers, including knowing about implementation constraints imposed by some USB controllers or understanding that composite devices might happen to be built by integrating reusable components.

Note that the lifecycle above can be slightly different for OTG devices. Other than providing an additional OTG descriptor in each configuration, only the HNP-related differences are particularly visible to driver code. They involve reporting requirements during the `SET_CONFIGURATION` request, and the option to invoke HNP during some suspend callbacks. Also, SRP changes the semantics of `usb_gadget_wakeup` slightly.

USB 2.0 Chapter 9 Types and Constants

Gadget drivers rely on common USB structures and constants defined in the *linux/usb/ch9.h* header file, which is standard in Linux 2.6+ kernels. These are the same types and constants used by host side drivers (and usbcore).

Core Objects and Methods

These are declared in <linux/usb/gadget.h>, and are used by gadget drivers to interact with USB peripheral controller drivers.

struct **usb_request**
describes one i/o request

Definition

```
struct usb_request {
  void *buf;
  unsigned length;
  dma_addr_t dma;
  struct scatterlist      *sg;
  unsigned num_sgs;
  unsigned num_mapped_sgs;
  unsigned stream_id:16;
  unsigned no_interrupt:1;
  unsigned zero:1;
  unsigned short_not_ok:1;
  unsigned dma_mapped:1;
  void (*complete)(struct usb_ep *ep, struct usb_request *req);
  void *context;
  struct list_head        list;
  int status;
  unsigned actual;
};
```

Members

buf Buffer used for data. Always provide this; some controllers only use PIO, or don't use DMA for some endpoints.

length Length of that data

dma DMA address corresponding to 'buf'. If you don't set this field, and the usb controller needs one, it is responsible for mapping and unmapping the buffer.

sg a scatterlist for SG-capable controllers.

num_sgs number of SG entries

num_mapped_sgs number of SG entries mapped to DMA (internal)

stream_id The stream id, when USB3.0 bulk streams are being used

no_interrupt If true, hints that no completion irq is needed. Helpful sometimes with deep request queues that are handled directly by DMA controllers.

zero If true, when writing data, makes the last packet be "short" by adding a zero length packet as needed;

short_not_ok When reading data, makes short packets be treated as errors (queue stops advancing till cleanup).

dma_mapped Indicates if request has been mapped to DMA (internal)

complete Function called when request completes, so this request and its buffer may be re-used. The function will always be called with interrupts disabled, and it must not sleep. Reads terminate with a short packet, or when the buffer fills, whichever comes first. When writes terminate, some data bytes will usually still be in flight (often in a hardware fifo). Errors (for reads or writes) stop the queue from advancing until the completion function returns, so that any transfers invalidated by the error may first be dequeued.

context For use by the completion callback

list For use by the gadget driver.

status Reports completion code, zero or a negative errno. Normally, faults block the transfer queue from advancing until the completion callback returns. Code "-ESHUTDOWN" indicates completion caused by device disconnect, or when the driver disabled the endpoint.

actual Reports bytes transferred to/from the buffer. For reads (OUT transfers) this may be less than the requested length. If the short_not_ok flag is set, short reads are treated as errors even when status otherwise indicates successful completion. Note that for writes (IN transfers) some data bytes may still reside in a device-side FIFO when the request is reported as complete.

Description

These are allocated/freed through the endpoint they're used with. The hardware's driver can add extra per-request data to the memory it returns, which often avoids separate memory allocations (potential failures), later when the request is queued.

Request flags affect request handling, such as whether a zero length packet is written (the "zero" flag), whether a short read should be treated as an error (blocking request queue advance, the "short_not_ok" flag), or hinting that an interrupt is not required (the "no_interrupt" flag, for use with deep request queues).

Bulk endpoints can use any size buffers, and can also be used for interrupt transfers. interrupt-only endpoints can be much less functional.

NOTE

this is analogous to 'struct urb' on the host side, except that it's thinner and promotes more pre-allocation.

struct **usb_ep_caps**
endpoint capabilities description

Definition

```
struct usb_ep_caps {
  unsigned type_control:1;
  unsigned type_iso:1;
  unsigned type_bulk:1;
  unsigned type_int:1;
  unsigned dir_in:1;
  unsigned dir_out:1;
};
```

Members

type_control Endpoint supports control type (reserved for ep0).

type_iso Endpoint supports isochronous transfers.

type_bulk Endpoint supports bulk transfers.

type_int Endpoint supports interrupt transfers.

dir_in Endpoint supports IN direction.

dir_out Endpoint supports OUT direction.

struct **usb_ep**
device side representation of USB endpoint

Definition

```
struct usb_ep {
  void *driver_data;
  const char              *name;
  const struct usb_ep_ops *ops;
  struct list_head        ep_list;
  struct usb_ep_caps      caps;
  bool claimed;
  bool enabled;
  unsigned maxpacket:16;
  unsigned maxpacket_limit:16;
  unsigned max_streams:16;
  unsigned mult:2;
  unsigned maxburst:5;
  u8 address;
  const struct usb_endpoint_descriptor    *desc;
  const struct usb_ss_ep_comp_descriptor  *comp_desc;
};
```

Members

`driver_data` for use by the gadget driver.

`name` identifier for the endpoint, such as "ep-a" or "ep9in-bulk"

`ops` Function pointers used to access hardware-specific operations.

`ep_list` the gadget's ep_list holds all of its endpoints

`caps` The structure describing types and directions supported by endoint.

`claimed` True if this endpoint is claimed by a function.

`enabled` The current endpoint enabled/disabled state.

`maxpacket` The maximum packet size used on this endpoint. The initial value can sometimes be reduced (hardware allowing), according to the endpoint descriptor used to configure the endpoint.

`maxpacket_limit` The maximum packet size value which can be handled by this endpoint. It's set once by UDC driver when endpoint is initialized, and should not be changed. Should not be confused with maxpacket.

`max_streams` The maximum number of streams supported by this EP (0 - 16, actual number is 2^n)

`mult` multiplier, 'mult' value for SS Isoc EPs

`maxburst` the maximum number of bursts supported by this EP (for usb3)

`address` used to identify the endpoint when finding descriptor that matches connection speed

`desc` endpoint descriptor. This pointer is set before the endpoint is enabled and remains valid until the endpoint is disabled.

`comp_desc` In case of SuperSpeed support, this is the endpoint companion descriptor that is used to configure the endpoint

Description

the bus controller driver lists all the general purpose endpoints in gadget->ep_list. the control endpoint (gadget->ep0) is not in that list, and is accessed only in response to a driver `setup()` callback.

struct **`usb_gadget`**
represents a usb slave device

Definition

```
struct usb_gadget {
  struct work_struct              work;
  struct usb_udc                  *udc;
  const struct usb_gadget_ops     *ops;
  struct usb_ep                   *ep0;
  struct list_head                ep_list;
  enum usb_device_speed           speed;
  enum usb_device_speed           max_speed;
  enum usb_device_state           state;
  const char                      *name;
  struct device                   dev;
  unsigned isoch_delay;
  unsigned out_epnum;
  unsigned in_epnum;
  unsigned mA;
  struct usb_otg_caps            *otg_caps;
  unsigned sg_supported:1;
  unsigned is_otg:1;
  unsigned is_a_peripheral:1;
  unsigned b_hnp_enable:1;
  unsigned a_hnp_support:1;
  unsigned a_alt_hnp_support:1;
  unsigned hnp_polling_support:1;
  unsigned host_request_flag:1;
  unsigned quirk_ep_out_aligned_size:1;
  unsigned quirk_altset_not_supp:1;
  unsigned quirk_stall_not_supp:1;
  unsigned quirk_zlp_not_supp:1;
  unsigned quirk_avoids_skb_reserve:1;
  unsigned is_selfpowered:1;
  unsigned deactivated:1;
  unsigned connected:1;
  unsigned lpm_capable:1;
};
```

Members

`work` (internal use) Workqueue to be used for `sysfs_notify()`

`udc` struct usb_udc pointer for this gadget

`ops` Function pointers used to access hardware-specific operations.

`ep0` Endpoint zero, used when reading or writing responses to driver `setup()` requests

`ep_list` List of other endpoints supported by the device.

`speed` Speed of current connection to USB host.

`max_speed` Maximal speed the UDC can handle. UDC must support this and all slower speeds.

`state` the state we are now (attached, suspended, configured, etc)

`name` Identifies the controller hardware type. Used in diagnostics and sometimes configuration.

`dev` Driver model state for this abstract device.

`isoch_delay` value from Set Isoch Delay request. Only valid on SS/SSP

`out_epnum` last used out ep number

`in_epnum` last used in ep number

`mA` last set mA value

`otg_caps` OTG capabilities of this gadget.

`sg_supported` true if we can handle scatter-gather

`is_otg` True if the USB device port uses a Mini-AB jack, so that the gadget driver must provide a USB OTG descriptor.

`is_a_peripheral` False unless is_otg, the "A" end of a USB cable is in the Mini-AB jack, and HNP has been used to switch roles so that the "A" device currently acts as A-Peripheral, not A-Host.

`b_hnp_enable` OTG device feature flag, indicating that the A-Host enabled HNP support.

`a_hnp_support` OTG device feature flag, indicating that the A-Host supports HNP at this port.

`a_alt_hnp_support` OTG device feature flag, indicating that the A-Host only supports HNP on a different root port.

hnp_polling_support OTG device feature flag, indicating if the OTG device in peripheral mode can support HNP polling.

host_request_flag OTG device feature flag, indicating if A-Peripheral or B-Peripheral wants to take host role.

quirk_ep_out_aligned_size epout requires buffer size to be aligned to MaxPacketSize.

quirk_altset_not_supp UDC controller doesn't support alt settings.

quirk_stall_not_supp UDC controller doesn't support stalling.

quirk_zlp_not_supp UDC controller doesn't support ZLP.

quirk_avoids_skb_reserve udc/platform wants to avoid `skb_reserve()` in u_ether.c to improve performance.

is_selfpowered if the gadget is self-powered.

deactivated True if gadget is deactivated - in deactivated state it cannot be connected.

connected True if gadget is connected.

lpm_capable If the gadget max_speed is FULL or HIGH, this flag indicates that it supports LPM as per the LPM ECN & errata.

Description

Gadgets have a mostly-portable "gadget driver" implementing device functions, handling all usb configurations and interfaces. Gadget drivers talk to hardware-specific code indirectly, through ops vectors. That insulates the gadget driver from hardware details, and packages the hardware endpoints through generic i/o queues. The "usb_gadget" and "usb_ep" interfaces provide that insulation from the hardware.

Except for the driver data, all fields in this structure are read-only to the gadget driver. That driver data is part of the "driver model" infrastructure in 2.6 (and later) kernels, and for earlier systems is grouped in a similar structure that's not known to the rest of the kernel.

Values of the three OTG device feature flags are updated before the `setup()` call corresponding to USB_REQ_SET_CONFIGURATION, and before driver `suspend()` calls. They are valid only when is_otg, and when the device is acting as a B-Peripheral (so is_a_peripheral is false).

size_t **usb_ep_align**(struct *usb_ep* * *ep*, size_t *len*)
returns **len** aligned to ep's maxpacketsize.

Parameters

struct usb_ep * ep the endpoint whose maxpacketsize is used to align **len**

size_t len buffer size's length to align to **ep**'s maxpacketsize

Description

This helper is used to align buffer's size to an ep's maxpacketsize.

size_t **usb_ep_align_maybe**(struct *usb_gadget* * *g*, struct *usb_ep* * *ep*, size_t *len*)
returns **len** aligned to ep's maxpacketsize if gadget requires quirk_ep_out_aligned_size, otherwise returns len.

Parameters

struct usb_gadget * g controller to check for quirk

struct usb_ep * ep the endpoint whose maxpacketsize is used to align **len**

size_t len buffer size's length to align to **ep**'s maxpacketsize

Description

This helper is used in case it's required for any reason to check and maybe align buffer's size to an ep's maxpacketsize.

int **gadget_is_altset_supported**(struct *usb_gadget* * *g*)
return true iff the hardware supports altsettings

Parameters

struct usb_gadget * g controller to check for quirk

int **gadget_is_stall_supported**(struct *usb_gadget* * *g*)
return true iff the hardware supports stalling

Parameters

struct usb_gadget * g controller to check for quirk

int **gadget_is_zlp_supported**(struct *usb_gadget* * *g*)
return true iff the hardware supports zlp

Parameters

struct usb_gadget * g controller to check for quirk

int **gadget_avoids_skb_reserve**(struct *usb_gadget* * *g*)
return true iff the hardware would like to avoid skb_reserve to improve performance.

Parameters

struct usb_gadget * g controller to check for quirk

int **gadget_is_dualspeed**(struct *usb_gadget* * *g*)
return true iff the hardware handles high speed

Parameters

struct usb_gadget * g controller that might support both high and full speeds

int **gadget_is_superspeed**(struct *usb_gadget* * *g*)
return true if the hardware handles superspeed

Parameters

struct usb_gadget * g controller that might support superspeed

int **gadget_is_superspeed_plus**(struct *usb_gadget* * *g*)
return true if the hardware handles superspeed plus

Parameters

struct usb_gadget * g controller that might support superspeed plus

int **gadget_is_otg**(struct *usb_gadget* * *g*)
return true iff the hardware is OTG-ready

Parameters

struct usb_gadget * g controller that might have a Mini-AB connector

Description

This is a runtime test, since kernels with a USB-OTG stack sometimes run on boards which only have a Mini-B (or Mini-A) connector.

struct **usb_gadget_driver**
driver for usb 'slave' devices

Definition

```
struct usb_gadget_driver {
  char *function;
  enum usb_device_speed   max_speed;
  int (*bind)(struct usb_gadget *gadget, struct usb_gadget_driver *driver);
  void (*unbind)(struct usb_gadget *);
  int (*setup)(struct usb_gadget *, const struct usb_ctrlrequest *);
  void (*disconnect)(struct usb_gadget *);
  void (*suspend)(struct usb_gadget *);
  void (*resume)(struct usb_gadget *);
  void (*reset)(struct usb_gadget *);
  struct device_driver    driver;
  char *udc_name;
  struct list_head        pending;
  unsigned match_existing_only:1;
};
```

Members

`function` String describing the gadget's function

`max_speed` Highest speed the driver handles.

`bind` the driver's bind callback

`unbind` Invoked when the driver is unbound from a gadget, usually from rmmod (after a disconnect is reported). Called in a context that permits sleeping.

`setup` Invoked for ep0 control requests that aren't handled by the hardware level driver. Most calls must be handled by the gadget driver, including descriptor and configuration management. The 16 bit members of the setup data are in USB byte order. Called in_interrupt; this may not sleep. Driver queues a response to ep0, or returns negative to stall.

`disconnect` Invoked after all transfers have been stopped, when the host is disconnected. May be called in_interrupt; this may not sleep. Some devices can't detect disconnect, so this might not be called except as part of controller shutdown.

`suspend` Invoked on USB suspend. May be called in_interrupt.

`resume` Invoked on USB resume. May be called in_interrupt.

`reset` Invoked on USB bus reset. It is mandatory for all gadget drivers and should be called in_interrupt.

`driver` Driver model state for this driver.

`udc_name` A name of UDC this driver should be bound to. If udc_name is NULL, this driver will be bound to any available UDC.

`pending` UDC core private data used for deferred probe of this driver.

`match_existing_only` If udc is not found, return an error and don't add this gadget driver to list of pending driver

Description

Devices are disabled till a gadget driver successfully `` bind()`s, which means the driver will handle :c:func:`setup() `` requests needed to enumerate (and meet "chapter 9" requirements) then do some useful work.

If gadget->is_otg is true, the gadget driver must provide an OTG descriptor during enumeration, or else fail the `bind()` call. In such cases, no USB traffic may flow until both `bind()` returns without having called `usb_gadget_disconnect()`, and the USB host stack has initialized.

Drivers use hardware-specific knowledge to configure the usb hardware. endpoint addressing is only one of several hardware characteristics that are in descriptors the ep0 implementation returns from `setup()` calls.

Except for ep0 implementation, most driver code shouldn't need change to run on top of different usb controllers. It'll use endpoints set up by that ep0 implementation.

The usb controller driver handles a few standard usb requests. Those include set_address, and feature flags for devices, interfaces, and endpoints (the get_status, set_feature, and clear_feature requests).

Accordingly, the driver's `setup()` callback must always implement all get_descriptor requests, returning at least a device descriptor and a configuration descriptor. Drivers must make sure the endpoint descriptors match any hardware constraints. Some hardware also constrains other descriptors. (The pxa250 allows only configurations 1, 2, or 3).

The driver's `setup()` callback must also implement set_configuration, and should also implement set_interface, get_configuration, and get_interface. Setting a configuration (or interface) is where endpoints should be activated or (config 0) shut down.

(Note that only the default control endpoint is supported. Neither hosts nor devices generally support control traffic except to ep0.)

Most devices will ignore USB suspend/resume operations, and so will not provide those callbacks. However, some may need to change modes when the host is not longer directing those activities. For example, local controls (buttons, dials, etc) may need to be re-enabled since the (remote) host can't do that any longer; or an error state might be cleared, to make the device behave identically whether or not power is maintained.

int **`usb_gadget_probe_driver`**(struct *usb_gadget_driver* * *driver*)
probe a gadget driver

Parameters

`struct usb_gadget_driver * driver` the driver being registered

Context

can sleep

Description

Call this in your gadget driver's module initialization function, to tell the underlying usb controller driver about your driver. The **bind()** function will be called to bind it to a gadget before this registration call returns. It's expected that the **bind()** function will be in init sections.

int **`usb_gadget_unregister_driver`**(struct *usb_gadget_driver* * *driver*)
unregister a gadget driver

Parameters

`struct usb_gadget_driver * driver` the driver being unregistered

Context

can sleep

Description

Call this in your gadget driver's module cleanup function, to tell the underlying usb controller that your driver is going away. If the controller is connected to a USB host, it will first `disconnect()`. The driver is also requested to `unbind()` and clean up any device state, before this procedure finally returns. It's expected that the `unbind()` functions will in in exit sections, so may not be linked in some kernels.

struct **`usb_string`**
wraps a C string and its USB id

Definition

```
struct usb_string {
  u8 id;
  const char              *s;
};
```

Members

`id` the (nonzero) ID for this string

`s` the string, in UTF-8 encoding

Description

If you're using *`usb_gadget_get_string()`*, use this to wrap a string together with its ID.

struct **usb_gadget_strings**
a set of USB strings in a given language

Definition

```
struct usb_gadget_strings {
  u16 language;
  struct usb_string       *strings;
};
```

Members

language identifies the strings' language (0x0409 for en-us)

strings array of strings with their ids

Description

If you're using *usb_gadget_get_string()*, use this to wrap all the strings for a given language.

void **usb_free_descriptors**(struct usb_descriptor_header ** *v*)
free descriptors returned by *usb_copy_descriptors()*

Parameters

struct usb_descriptor_header ** v vector of descriptors

Optional Utilities

The core API is sufficient for writing a USB Gadget Driver, but some optional utilities are provided to simplify common tasks. These utilities include endpoint autoconfiguration.

int **usb_gadget_get_string**(struct *usb_gadget_strings* * *table*, int *id*, u8 * *buf*)
fill out a string descriptor

Parameters

struct usb_gadget_strings * table of c strings encoded using UTF-8

int id string id, from low byte of wValue in get string descriptor

u8 * buf at least 256 bytes, must be 16-bit aligned

Description

Finds the UTF-8 string matching the ID, and converts it into a string descriptor in utf16-le. Returns length of descriptor (always even) or negative errno

If your driver needs stings in multiple languages, you'll probably "switch (wIndex) { … }" in your ep0 string descriptor logic, using this routine after choosing which set of UTF-8 strings to use. Note that US-ASCII is a strict subset of UTF-8; any string bytes with the eighth bit set will be multibyte UTF-8 characters, not ISO-8859/1 characters (which are also widely used in C strings).

int **usb_descriptor_fillbuf**(void * *buf*, unsigned *buflen*, const struct usb_descriptor_header ** *src*)
fill buffer with descriptors

Parameters

void * buf Buffer to be filled

unsigned buflen Size of buf

const struct usb_descriptor_header ** src Array of descriptor pointers, terminated by null pointer.

Description

Copies descriptors into the buffer, returning the length or a negative error code if they can't all be copied. Useful when assembling descriptors for an associated set of interfaces used as part of configuring a composite device; or in other cases where sets of descriptors need to be marshaled.

int **usb_gadget_config_buf**(const struct usb_config_descriptor * *config*, void * *buf*, unsigned *length*, const struct usb_descriptor_header ** *desc*)
builts a complete configuration descriptor

Parameters

const struct usb_config_descriptor * config Header for the descriptor, including characteristics such as power requirements and number of interfaces.

void * buf Buffer for the resulting configuration descriptor.

unsigned length Length of buffer. If this is not big enough to hold the entire configuration descriptor, an error code will be returned.

const struct usb_descriptor_header ** desc Null-terminated vector of pointers to the descriptors (interface, endpoint, etc) defining all functions in this device configuration.

Description

This copies descriptors into the response buffer, building a descriptor for that configuration. It returns the buffer length or a negative status code. The config.wTotalLength field is set to match the length of the result, but other descriptor fields (including power usage and interface count) must be set by the caller.

Gadget drivers could use this when constructing a config descriptor in response to USB_REQ_GET_DESCRIPTOR. They will need to patch the resulting bDescriptorType value if USB_DT_OTHER_SPEED_CONFIG is needed.

struct usb_descriptor_header ** **usb_copy_descriptors**(struct usb_descriptor_header ** *src*)
copy a vector of USB descriptors

Parameters

struct usb_descriptor_header ** src null-terminated vector to copy

Context

initialization code, which may sleep

Description

This makes a copy of a vector of USB descriptors. Its primary use is to support usb_function objects which can have multiple copies, each needing different descriptors. Functions may have static tables of descriptors, which are used as templates and customized with identifiers (for interfaces, strings, endpoints, and more) as needed by a given function instance.

Composite Device Framework

The core API is sufficient for writing drivers for composite USB devices (with more than one function in a given configuration), and also multi-configuration devices (also more than one function, but not necessarily sharing a given configuration). There is however an optional framework which makes it easier to reuse and combine functions.

Devices using this framework provide a struct *usb_composite_driver*, which in turn provides one or more struct *usb_configuration* instances. Each such configuration includes at least one struct *usb_function*, which packages a user visible role such as "network link" or "mass storage device". Management functions may also exist, such as "Device Firmware Upgrade".

struct **usb_os_desc_ext_prop**
describes one "Extended Property"

Definition

```
struct usb_os_desc_ext_prop {
  struct list_head        entry;
  u8 type;
  int name_len;
  char *name;
  int data_len;
  char *data;
  struct config_item      item;
};
```

Members

entry used to keep a list of extended properties

type Extended Property type

name_len Extended Property unicode name length, including terminating ‘0’

name Extended Property name

data_len Length of Extended Property blob (for unicode store double len)

data Extended Property blob

item Represents this Extended Property in configfs

struct **usb_os_desc**
describes OS descriptors associated with one interface

Definition

```
struct usb_os_desc {
  char *ext_compat_id;
  struct list_head        ext_prop;
  int ext_prop_len;
  int ext_prop_count;
  struct mutex            *opts_mutex;
  struct config_group     group;
  struct module           *owner;
};
```

Members

ext_compat_id 16 bytes of “Compatible ID” and “Subcompatible ID”

ext_prop Extended Properties list

ext_prop_len Total length of Extended Properties blobs

ext_prop_count Number of Extended Properties

opts_mutex Optional mutex protecting config data of a usb_function_instance

group Represents OS descriptors associated with an interface in configfs

owner Module associated with this OS descriptor

struct **usb_os_desc_table**
describes OS descriptors associated with one interface of a usb_function

Definition

```
struct usb_os_desc_table {
  int if_id;
  struct usb_os_desc      *os_desc;
};
```

Members

if_id Interface id

os_desc “Extended Compatibility ID” and “Extended Properties” of the interface

Description

Each interface can have at most one "Extended Compatibility ID" and a number of "Extended Properties".

struct **usb_function**
describes one function of a configuration

Definition

```
struct usb_function {
  const char                     *name;
  struct usb_gadget_strings      **strings;
  struct usb_descriptor_header    **fs_descriptors;
  struct usb_descriptor_header    **hs_descriptors;
  struct usb_descriptor_header    **ss_descriptors;
  struct usb_descriptor_header    **ssp_descriptors;
  struct usb_configuration       *config;
  struct usb_os_desc_table       *os_desc_table;
  unsigned os_desc_n;
  int (*bind)(struct usb_configuration *, struct usb_function *);
  void (*unbind)(struct usb_configuration *, struct usb_function *);
  void (*free_func)(struct usb_function *f);
  struct module           *mod;
  int (*set_alt)(struct usb_function *, unsigned interface, unsigned alt);
  int (*get_alt)(struct usb_function *, unsigned interface);
  void (*disable)(struct usb_function *);
  int (*setup)(struct usb_function *, const struct usb_ctrlrequest *);
  bool (*req_match)(struct usb_function *,const struct usb_ctrlrequest *, bool config0);
  void (*suspend)(struct usb_function *);
  void (*resume)(struct usb_function *);
  int (*get_status)(struct usb_function *);
  int (*func_suspend)(struct usb_function *, u8 suspend_opt);
};
```

Members

name For diagnostics, identifies the function.

strings tables of strings, keyed by identifiers assigned during bind() and by language IDs provided in control requests

fs_descriptors Table of full (or low) speed descriptors, using interface and string identifiers assigned during **bind()**. If this pointer is null, the function will not be available at full speed (or at low speed).

hs_descriptors Table of high speed descriptors, using interface and string identifiers assigned during **bind()**. If this pointer is null, the function will not be available at high speed.

ss_descriptors Table of super speed descriptors, using interface and string identifiers assigned during **bind()**. If this pointer is null after initiation, the function will not be available at super speed.

ssp_descriptors Table of super speed plus descriptors, using interface and string identifiers assigned during **bind()**. If this pointer is null after initiation, the function will not be available at super speed plus.

config assigned when **usb_add_function()** is called; this is the configuration with which this function is associated.

os_desc_table Table of (interface id, os descriptors) pairs. The function can expose more than one interface. If an interface is a member of an IAD, only the first interface of IAD has its entry in the table.

os_desc_n Number of entries in os_desc_table

bind Before the gadget can register, all of its functions bind() to the available resources including string and interface identifiers used in interface or class descriptors; endpoints; I/O buffers; and so on.

unbind Reverses **bind**; called as a side effect of unregistering the driver which added this function.

free_func free the struct usb_function.

mod (internal) points to the module that created this structure.

set_alt (REQUIRED) Reconfigures altsettings; function drivers may initialize usb_ep.driver data at this time (when it is used). Note that setting an interface to its current altsetting resets interface state, and that all interfaces have a disabled state.

get_alt Returns the active altsetting. If this is not provided, then only altsetting zero is supported.

disable (REQUIRED) Indicates the function should be disabled. Reasons include host resetting or reconfiguring the gadget, and disconnection.

setup Used for interface-specific control requests.

req_match Tests if a given class request can be handled by this function.

suspend Notifies functions when the host stops sending USB traffic.

resume Notifies functions when the host restarts USB traffic.

get_status Returns function status as a reply to `GetStatus()` request when the recipient is Interface.

func_suspend callback to be called when SetFeature(FUNCTION_SUSPEND) is reseived

Description

A single USB function uses one or more interfaces, and should in most cases support operation at both full and high speeds. Each function is associated by **usb_add_function()** with a one configuration; that function causes **bind()** to be called so resources can be allocated as part of setting up a gadget driver. Those resources include endpoints, which should be allocated using **usb_ep_autoconfig()**.

To support dual speed operation, a function driver provides descriptors for both high and full speed operation. Except in rare cases that don't involve bulk endpoints, each speed needs different endpoint descriptors.

Function drivers choose their own strategies for managing instance data. The simplest strategy just declares it "static', which means the function can only be activated once. If the function needs to be exposed in more than one configuration at a given speed, it needs to support multiple usb_function structures (one for each configuration).

A more complex strategy might encapsulate a **usb_function** structure inside a driver-specific instance structure to allows multiple activations. An example of multiple activations might be a CDC ACM function that supports two or more distinct instances within the same configuration, providing several independent logical data links to a USB host.

struct **usb_configuration**
    represents one gadget configuration

Definition

```
struct usb_configuration {
  const char                      *label;
  struct usb_gadget_strings       **strings;
  const struct usb_descriptor_header **descriptors;
  void (*unbind)(struct usb_configuration *);
  int (*setup)(struct usb_configuration *, const struct usb_ctrlrequest *);
  u8 bConfigurationValue;
  u8 iConfiguration;
  u8 bmAttributes;
  u16 MaxPower;
  struct usb_composite_dev        *cdev;
};
```

Members

label For diagnostics, describes the configuration.

strings Tables of strings, keyed by identifiers assigned during **bind()** and by language IDs provided in control requests.

descriptors Table of descriptors preceding all function descriptors. Examples include OTG and vendor-specific descriptors.

unbind Reverses **bind**; called as a side effect of unregistering the driver which added this configuration.

setup Used to delegate control requests that aren't handled by standard device infrastructure or directed at a specific interface.

bConfigurationValue Copied into configuration descriptor.

iConfiguration Copied into configuration descriptor.

bmAttributes Copied into configuration descriptor.

MaxPower Power consumtion in mA. Used to compute bMaxPower in the configuration descriptor after considering the bus speed.

cdev assigned by **usb_add_config()** before calling **bind()**; this is the device associated with this configuration.

Description

Configurations are building blocks for gadget drivers structured around function drivers. Simple USB gadgets require only one function and one configuration, and handle dual-speed hardware by always providing the same functionality. Slightly more complex gadgets may have more than one single-function configuration at a given speed; or have configurations that only work at one speed.

Composite devices are, by definition, ones with configurations which include more than one function.

The lifecycle of a usb_configuration includes allocation, initialization of the fields described above, and calling **usb_add_config()** to set up internal data and bind it to a specific device. The configuration's **bind()** method is then used to initialize all the functions and then call **usb_add_function()** for them.

Those functions would normally be independent of each other, but that's not mandatory. CDC WMC devices are an example where functions often depend on other functions, with some functions subsidiary to others. Such interdependency may be managed in any way, so long as all of the descriptors complete by the time the composite driver returns from its `bind()` routine.

struct **usb_composite_driver**
groups configurations into a gadget

Definition

```
struct usb_composite_driver {
  const char                              *name;
  const struct usb_device_descriptor      *dev;
  struct usb_gadget_strings               **strings;
  enum usb_device_speed                   max_speed;
  unsigned needs_serial:1;
  int (*bind)(struct usb_composite_dev *cdev);
  int (*unbind)(struct usb_composite_dev *);
  void (*disconnect)(struct usb_composite_dev *);
  void (*suspend)(struct usb_composite_dev *);
  void (*resume)(struct usb_composite_dev *);
  struct usb_gadget_driver                gadget_driver;
};
```

Members

name For diagnostics, identifies the driver.

dev Template descriptor for the device, including default device identifiers.

strings tables of strings, keyed by identifiers assigned during **bind** and language IDs provided in control requests. Note: The first entries are predefined. The first entry that may be used is USB_GADGET_FIRST_AVAIL_IDX

max_speed Highest speed the driver supports.

needs_serial set to 1 if the gadget needs userspace to provide a serial number. If one is not provided, warning will be printed.

bind (REQUIRED) Used to allocate resources that are shared across the whole device, such as string IDs, and add its configurations using **usb_add_config()**. This may fail by returning a negative errno value; it should return zero on successful initialization.

unbind Reverses **bind**; called as a side effect of unregistering this driver.

disconnect optional driver disconnect method

suspend Notifies when the host stops sending USB traffic, after function notifications

resume Notifies configuration when the host restarts USB traffic, before function notifications

gadget_driver Gadget driver controlling this driver

Description

Devices default to reporting self powered operation. Devices which rely on bus powered operation should report this in their **bind** method.

Before returning from **bind**, various fields in the template descriptor may be overridden. These include the idVendor/idProduct/bcdDevice values normally to bind the appropriate host side driver, and the three strings (iManufacturer, iProduct, iSerialNumber) normally used to provide user meaningful device identifiers. (The strings will not be defined unless they are defined in **dev** and **strings**.) The correct ep0 maxpacket size is also reported, as defined by the underlying controller driver.

module_usb_composite_driver(*__usb_composite_driver*)
Helper macro for registering a USB gadget composite driver

Parameters

__usb_composite_driver usb_composite_driver struct

Description

Helper macro for USB gadget composite 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()*

struct **usb_composite_dev**
represents one composite usb gadget

Definition

```
struct usb_composite_dev {
  struct usb_gadget              *gadget;
  struct usb_request             *req;
  struct usb_request             *os_desc_req;
  struct usb_configuration       *config;
  u8 qw_sign[OS_STRING_QW_SIGN_LEN];
  u8 b_vendor_code;
  struct usb_configuration       *os_desc_config;
  unsigned int                   use_os_string:1;
  unsigned int                   setup_pending:1;
  unsigned int                   os_desc_pending:1;
};
```

Members

gadget read-only, abstracts the gadget's usb peripheral controller

req used for control responses; buffer is pre-allocated

os_desc_req used for OS descriptors responses; buffer is pre-allocated

config the currently active configuration

qw_sign qwSignature part of the OS string

b_vendor_code bMS_VendorCode part of the OS string

os_desc_config the configuration to be used with OS descriptors

use_os_string false by default, interested gadgets set it

setup_pending true when setup request is queued but not completed

os_desc_pending true when os_desc request is queued but not completed

Description

One of these devices is allocated and initialized before the associated device driver's `bind()` is called.

OPEN ISSUE: it appears that some WUSB devices will need to be built by combining a normal (wired) gadget with a wireless one. This revision of the gadget framework should probably try to make sure doing that won't hurt too much.

One notion for how to handle Wireless USB devices involves:

1. a second gadget here, discovery mechanism TBD, but likely needing separate "register/unregister WUSB gadget" calls;
2. updates to usb_gadget to include flags "is it wireless", "is it wired", plus (presumably in a wrapper structure) bandgroup and PHY info;
3. presumably a wireless_ep wrapping a usb_ep, and reporting wireless-specific parameters like maxburst and maxsequence;
4. configurations that are specific to wireless links;
5. function drivers that understand wireless configs and will support wireless for (additional) function instances;
6. a function to support association setup (like CBAF), not necessarily requiring a wireless adapter;
7. composite device setup that can create one or more wireless configs, including appropriate association setup support;
8. more, TBD.

int **config_ep_by_speed**(struct *usb_gadget* * *g*, struct *usb_function* * *f*, struct *usb_ep* * *_ep*)
configures the given endpoint according to gadget speed.

Parameters

struct usb_gadget * g pointer to the gadget

struct usb_function * f usb function

struct usb_ep * _ep the endpoint to configure

Return

error code, 0 on success

This function chooses the right descriptors for a given endpoint according to gadget speed and saves it in the endpoint desc field. If the endpoint already has a descriptor assigned to it - overwrites it with currently corresponding descriptor. The endpoint maxpacket field is updated according to the chosen descriptor.

Note

the supplied function should hold all the descriptors for supported speeds

int **usb_add_function**(struct *usb_configuration* * *config*, struct *usb_function* * *function*)
add a function to a configuration

Parameters

struct usb_configuration * config the configuration

struct usb_function * function the function being added

Context

single threaded during gadget setup

Description

After initialization, each configuration must have one or more functions added to it. Adding a function involves calling its **bind()** method to allocate resources such as interface and string identifiers and endpoints.

This function returns the value of the function's `bind()`, which is zero for success else a negative errno value.

int **`usb_function_deactivate`**(struct *usb_function* * *function*)
prevent function and gadget enumeration

Parameters

`struct usb_function * function` the function that isn't yet ready to respond

Description

Blocks response of the gadget driver to host enumeration by preventing the data line pullup from being activated. This is normally called during **bind()** processing to change from the initial "ready to respond" state, or when a required resource becomes available.

For example, drivers that serve as a passthrough to a userspace daemon can block enumeration unless that daemon (such as an OBEX, MTP, or print server) is ready to handle host requests.

Not all systems support software control of their USB peripheral data pullups.

Returns zero on success, else negative errno.

int **`usb_function_activate`**(struct *usb_function* * *function*)
allow function and gadget enumeration

Parameters

`struct usb_function * function` function on which *`usb_function_activate()`* was called

Description

Reverses effect of *`usb_function_deactivate()`*. If no more functions are delaying their activation, the gadget driver will respond to host enumeration procedures.

Returns zero on success, else negative errno.

int **`usb_interface_id`**(struct *usb_configuration* * *config*, struct *usb_function* * *function*)
allocate an unused interface ID

Parameters

`struct usb_configuration * config` configuration associated with the interface

`struct usb_function * function` function handling the interface

Context

single threaded during gadget setup

Description

`usb_interface_id()` is called from usb_function.:c:func:*bind()* callbacks to allocate new interface IDs. The function driver will then store that ID in interface, association, CDC union, and other descriptors. It will also handle any control requests targeted at that interface, particularly changing its altsetting via `set_alt()`. There may also be class-specific or vendor-specific requests to handle.

All interface identifier should be allocated using this routine, to ensure that for example different functions don't wrongly assign different meanings to the same identifier. Note that since interface identifiers are configuration-specific, functions used in more than one configuration (or more than once in a given configuration) need multiple versions of the relevant descriptors.

Returns the interface ID which was allocated; or -ENODEV if no more interface IDs can be allocated.

int **usb_add_config**(struct *usb_composite_dev* * *cdev*, struct *usb_configuration* * *config*, int (*bind) (struct *usb_configuration* *)
add a configuration to a device.

Parameters

struct usb_composite_dev * cdev wraps the USB gadget

struct usb_configuration * config the configuration, with bConfigurationValue assigned

int (*)(struct usb_configuration *) bind the configuration's bind function

Context

single threaded during gadget setup

Description

One of the main tasks of a composite **bind()** routine is to add each of the configurations it supports, using this routine.

This function returns the value of the configuration's **bind()**, which is zero for success else a negative errno value. Binding configurations assigns global resources including string IDs, and per-configuration resources such as interface IDs and endpoints.

int **usb_string_id**(struct *usb_composite_dev* * *cdev*)
allocate an unused string ID

Parameters

struct usb_composite_dev * cdev the device whose string descriptor IDs are being allocated

Context

single threaded during gadget setup

Description

usb_string_id() is called from `bind()` callbacks to allocate string IDs. Drivers for functions, configurations, or gadgets will then store that ID in the appropriate descriptors and string table.

All string identifier should be allocated using this, **usb_string_ids_tab()** or **usb_string_ids_n()** routine, to ensure that for example different functions don't wrongly assign different meanings to the same identifier.

int **usb_string_ids_tab**(struct *usb_composite_dev* * *cdev*, struct *usb_string* * *str*)
allocate unused string IDs in batch

Parameters

struct usb_composite_dev * cdev the device whose string descriptor IDs are being allocated

struct usb_string * str an array of usb_string objects to assign numbers to

Context

single threaded during gadget setup

Description

usb_string_ids() is called from `bind()` callbacks to allocate string IDs. Drivers for functions, configurations, or gadgets will then copy IDs from the string table to the appropriate descriptors and string table for other languages.

All string identifier should be allocated using this, **usb_string_id()** or **usb_string_ids_n()** routine, to ensure that for example different functions don't wrongly assign different meanings to the same identifier.

struct *usb_string* * **usb_gstrings_attach**(struct *usb_composite_dev* * *cdev*, struct *usb_gadget_strings* ** *sp*, unsigned *n_strings*)
attach gadget strings to a cdev and assign ids

Parameters

struct usb_composite_dev * cdev the device whose string descriptor IDs are being allocated and attached.

struct usb_gadget_strings ** sp an array of usb_gadget_strings to attach.

unsigned n_strings number of entries in each usb_strings array (sp[]->strings)

Description

This function will create a deep copy of usb_gadget_strings and usb_string and attach it to the cdev. The actual string (usb_string.s) will not be copied but only a referenced will be made. The struct usb_gadget_strings array may contain multiple languages and should be NULL terminated. The ->language pointer of each struct usb_gadget_strings has to contain the same amount of entries. For instance: sp[0] is en-US, sp[1] is es-ES. It is expected that the first usb_string entry of es-ES contains the translation of the first usb_string entry of en-US. Therefore both entries become the same id assign.

int **usb_string_ids_n**(struct *usb_composite_dev* * *c*, unsigned *n*)
allocate unused string IDs in batch

Parameters

struct usb_composite_dev * c the device whose string descriptor IDs are being allocated

unsigned n number of string IDs to allocate

Context

single threaded during gadget setup

Description

Returns the first requested ID. This ID and next **n**-1 IDs are now valid IDs. At least provided that **n** is non-zero because if it is, returns last requested ID which is now very useful information.

usb_string_ids_n() is called from `bind()` callbacks to allocate string IDs. Drivers for functions, configurations, or gadgets will then store that ID in the appropriate descriptors and string table.

All string identifier should be allocated using this, **usb_string_id()** or **usb_string_ids_n()** routine, to ensure that for example different functions don't wrongly assign different meanings to the same identifier.

int **usb_composite_probe**(struct *usb_composite_driver* * *driver*)
register a composite driver

Parameters

struct usb_composite_driver * driver the driver to register

Context

single threaded during gadget setup

Description

This function is used to register drivers using the composite driver framework. The return value is zero, or a negative errno value. Those values normally come from the driver's **bind** method, which does all the work of setting up the driver to match the hardware.

On successful return, the gadget is ready to respond to requests from the host, unless one of its components invokes `usb_gadget_disconnect()` while it was binding. That would usually be done in order to wait for some userspace participation.

void **usb_composite_unregister**(struct *usb_composite_driver* * *driver*)
unregister a composite driver

Parameters

struct usb_composite_driver * driver the driver to unregister

Description

This function is used to unregister drivers using the composite driver framework.

void **usb_composite_setup_continue**(struct *usb_composite_dev* * *cdev*)
Continue with the control transfer

Parameters

struct usb_composite_dev * cdev the composite device who's control transfer was kept waiting

Description

This function must be called by the USB function driver to continue with the control transfer's data/status stage in case it had requested to delay the data/status stages. A USB function's setup handler (e.g. `set_alt()`) can request the composite framework to delay the setup request's data/status stages by returning USB_GADGET_DELAYED_STATUS.

Composite Device Functions

At this writing, a few of the current gadget drivers have been converted to this framework. Near-term plans include converting all of them, except for `gadgetfs`.

Peripheral Controller Drivers

The first hardware supporting this API was the NetChip 2280 controller, which supports USB 2.0 high speed and is based on PCI. This is the `net2280` driver module. The driver supports Linux kernel versions 2.4 and 2.6; contact NetChip Technologies for development boards and product information.

Other hardware working in the `gadget` framework includes: Intel's PXA 25x and IXP42x series processors (`pxa2xx_udc`), Toshiba TC86c001 "Goku-S" (`goku_udc`), Renesas SH7705/7727 (`sh_udc`), MediaQ 11xx (`mq11xx_udc`), Hynix HMS30C7202 (`h7202_udc`), National 9303/4 (`n9604_udc`), Texas Instruments OMAP (`omap_udc`), Sharp LH7A40x (`lh7a40x_udc`), and more. Most of those are full speed controllers.

At this writing, there are people at work on drivers in this framework for several other USB device controllers, with plans to make many of them be widely available.

A partial USB simulator, the `dummy_hcd` driver, is available. It can act like a net2280, a pxa25x, or an sa11x0 in terms of available endpoints and device speeds; and it simulates control, bulk, and to some extent interrupt transfers. That lets you develop some parts of a gadget driver on a normal PC, without any special hardware, and perhaps with the assistance of tools such as GDB running with User Mode Linux. At least one person has expressed interest in adapting that approach, hooking it up to a simulator for a microcontroller. Such simulators can help debug subsystems where the runtime hardware is unfriendly to software development, or is not yet available.

Support for other controllers is expected to be developed and contributed over time, as this driver framework evolves.

Gadget Drivers

In addition to *Gadget Zero* (used primarily for testing and development with drivers for usb controller hardware), other gadget drivers exist.

There's an `ethernet` gadget driver, which implements one of the most useful *Communications Device Class* (CDC) models. One of the standards for cable modem interoperability even specifies the use of this ethernet model as one of two mandatory options. Gadgets using this code look to a USB host as if they're an Ethernet adapter. It provides access to a network where the gadget's CPU is one host, which could easily be bridging, routing, or firewalling access to other networks. Since some hardware can't fully implement the CDC Ethernet requirements, this driver also implements a "good parts only" subset of CDC Ethernet. (That subset doesn't advertise itself as CDC Ethernet, to avoid creating problems.)

Support for Microsoft's `RNDIS` protocol has been contributed by Pengutronix and Auerswald GmbH. This is like CDC Ethernet, but it runs on more slightly USB hardware (but less than the CDC subset). However, its main claim to fame is being able to connect directly to recent versions of Windows, using drivers that Microsoft bundles and supports, making it much simpler to network with Windows.

There is also support for user mode gadget drivers, using `gadgetfs`. This provides a *User Mode API* that presents each endpoint as a single file descriptor. I/O is done using normal `read()` and `read()` calls. Familiar tools like GDB and pthreads can be used to develop and debug user mode drivers, so that once a robust controller driver is available many applications for it won't require new kernel mode software. Linux 2.6 *Async I/O (AIO)* support is available, so that user mode software can stream data with only slightly more overhead than a kernel driver.

There's a USB Mass Storage class driver, which provides a different solution for interoperability with systems such as MS-Windows and MacOS. That *Mass Storage* driver uses a file or block device as backing store for a drive, like the `loop` driver. The USB host uses the BBB, CB, or CBI versions of the mass storage class specification, using transparent SCSI commands to access the data from the backing store.

There's a "serial line" driver, useful for TTY style operation over USB. The latest version of that driver supports CDC ACM style operation, like a USB modem, and so on most hardware it can interoperate easily with MS-Windows. One interesting use of that driver is in boot firmware (like a BIOS), which can sometimes use that model with very small systems without real serial lines.

Support for other kinds of gadget is expected to be developed and contributed over time, as this driver framework evolves.

USB On-The-GO (OTG)

USB OTG support on Linux 2.6 was initially developed by Texas Instruments for OMAP 16xx and 17xx series processors. Other OTG systems should work in similar ways, but the hardware level details could be very different.

Systems need specialized hardware support to implement OTG, notably including a special *Mini-AB* jack and associated transceiver to support *Dual-Role* operation: they can act either as a host, using the standard Linux-USB host side driver stack, or as a peripheral, using this `gadget` framework. To do that, the system software relies on small additions to those programming interfaces, and on a new internal component (here called an "OTG Controller") affecting which driver stack connects to the OTG port. In each role, the system can re-use the existing pool of hardware-neutral drivers, layered on top of the controller driver interfaces (`usb_bus` or *`usb_gadget`*). Such drivers need at most minor changes, and most of the calls added to support OTG can also benefit non-OTG products.

- Gadget drivers test the `is_otg` flag, and use it to determine whether or not to include an OTG descriptor in each of their configurations.
- Gadget drivers may need changes to support the two new OTG protocols, exposed in new gadget attributes such as `b_hnp_enable` flag. HNP support should be reported through a user interface (two LEDs could suffice), and is triggered in some cases when the host suspends the peripheral. SRP support can be user-initiated just like remote wakeup, probably by pressing the same button.
- On the host side, USB device drivers need to be taught to trigger HNP at appropriate moments, using `usb_suspend_device()`. That also conserves battery power, which is useful even for non-OTG configurations.
- Also on the host side, a driver must support the OTG "Targeted Peripheral List". That's just a whitelist, used to reject peripherals not supported with a given Linux OTG host. *This whitelist is product-specific; each product must modify* `otg_whitelist.h` *to match its interoperability specification.*

  Non-OTG Linux hosts, like PCs and workstations, normally have some solution for adding drivers, so that peripherals that aren't recognized can eventually be supported. That approach is unreasonable for consumer products that may never have their firmware upgraded, and where it's usually unrealistic to expect traditional PC/workstation/server kinds of support model to work. For example, it's often impractical to change device firmware once the product has been distributed, so driver bugs can't normally be fixed if they're found after shipment.

Additional changes are needed below those hardware-neutral `usb_bus` and *usb_gadget* driver interfaces; those aren't discussed here in any detail. Those affect the hardware-specific code for each USB Host or Peripheral controller, and how the HCD initializes (since OTG can be active only on a single port). They also involve what may be called an *OTG Controller Driver*, managing the OTG transceiver and the OTG state machine logic as well as much of the root hub behavior for the OTG port. The OTG controller driver needs to activate and deactivate USB controllers depending on the relevant device role. Some related changes were needed inside usbcore, so that it can identify OTG-capable devices and respond appropriately to HNP or SRP protocols.

USB Anchors

What is anchor?

A USB driver needs to support some callbacks requiring a driver to cease all IO to an interface. To do so, a driver has to keep track of the URBs it has submitted to know they've all completed or to call usb_kill_urb for them. The anchor is a data structure takes care of keeping track of URBs and provides methods to deal with multiple URBs.

Allocation and Initialisation

There's no API to allocate an anchor. It is simply declared as struct usb_anchor. `init_usb_anchor()` must be called to initialise the data structure.

Deallocation

Once it has no more URBs associated with it, the anchor can be freed with normal memory management operations.

Association and disassociation of URBs with anchors

An association of URBs to an anchor is made by an explicit call to *usb_anchor_urb()*. The association is maintained until an URB is finished by (successful) completion. Thus disassociation is automatic. A function is provided to forcibly finish (kill) all URBs associated with an anchor. Furthermore, disassociation can be made with *usb_unanchor_urb()*

Operations on multitudes of URBs

`usb_kill_anchored_urbs()`

This function kills all URBs associated with an anchor. The URBs are called in the reverse temporal order they were submitted. This way no data can be reordered.

`usb_unlink_anchored_urbs()`

This function unlinks all URBs associated with an anchor. The URBs are processed in the reverse temporal order they were submitted. This is similar to *usb_kill_anchored_urbs()*, but it will not sleep. Therefore no guarantee is made that the URBs have been unlinked when the call returns. They may be unlinked later but will be unlinked in finite time.

usb_scuttle_anchored_urbs()

All URBs of an anchor are unanchored en masse.

usb_wait_anchor_empty_timeout()

This function waits for all URBs associated with an anchor to finish or a timeout, whichever comes first. Its return value will tell you whether the timeout was reached.

usb_anchor_empty()

Returns true if no URBs are associated with an anchor. Locking is the caller's responsibility.

usb_get_from_anchor()

Returns the oldest anchored URB of an anchor. The URB is unanchored and returned with a reference. As you may mix URBs to several destinations in one anchor you have no guarantee the chronologically first submitted URB is returned.

USB bulk streams

Background

Bulk endpoint streams were added in the USB 3.0 specification. Streams allow a device driver to overload a bulk endpoint so that multiple transfers can be queued at once.

Streams are defined in sections 4.4.6.4 and 8.12.1.4 of the Universal Serial Bus 3.0 specification at http://www.usb.org/developers/docs/ The USB Attached SCSI Protocol, which uses streams to queue multiple SCSI commands, can be found on the T10 website (http://t10.org/).

Device-side implications

Once a buffer has been queued to a stream ring, the device is notified (through an out-of-band mechanism on another endpoint) that data is ready for that stream ID. The device then tells the host which "stream" it wants to start. The host can also initiate a transfer on a stream without the device asking, but the device can refuse that transfer. Devices can switch between streams at any time.

Driver implications

```
int usb_alloc_streams(struct usb_interface *interface,
              struct usb_host_endpoint **eps, unsigned int num_eps,
              unsigned int num_streams, gfp_t mem_flags);
```

Device drivers will call this API to request that the host controller driver allocate memory so the driver can use up to num_streams stream IDs. They must pass an array of usb_host_endpoints that need to be setup with similar stream IDs. This is to ensure that a UASP driver will be able to use the same stream ID for the bulk IN and OUT endpoints used in a Bi-directional command sequence.

The return value is an error condition (if one of the endpoints doesn't support streams, or the xHCI driver ran out of memory), or the number of streams the host controller allocated for this endpoint. The xHCI host controller hardware declares how many stream IDs it can support, and each bulk endpoint on a SuperSpeed device will say how many stream IDs it can handle. Therefore, drivers should be able to deal with being allocated less stream IDs than they requested.

Do NOT call this function if you have URBs enqueued for any of the endpoints passed in as arguments. Do not call this function to request less than two streams.

Drivers will only be allowed to call this API once for the same endpoint without calling usb_free_streams(). This is a simplification for the xHCI host controller driver, and may change in the future.

Picking new Stream IDs to use

Stream ID 0 is reserved, and should not be used to communicate with devices. If usb_alloc_streams() returns with a value of N, you may use streams 1 though N. To queue an URB for a specific stream, set the urb->stream_id value. If the endpoint does not support streams, an error will be returned.

Note that new API to choose the next stream ID will have to be added if the xHCI driver supports secondary stream IDs.

Clean up

If a driver wishes to stop using streams to communicate with the device, it should call:

```
void usb_free_streams(struct usb_interface *interface,
              struct usb_host_endpoint **eps, unsigned int num_eps,
              gfp_t mem_flags);
```

All stream IDs will be deallocated when the driver releases the interface, to ensure that drivers that don't support streams will be able to use the endpoint.

USB core callbacks

What callbacks will usbcore do?

Usbcore will call into a driver through callbacks defined in the driver structure and through the completion handler of URBs a driver submits. Only the former are in the scope of this document. These two kinds of callbacks are completely independent of each other. Information on the completion callback can be found in *USB Request Block (URB)*.

The callbacks defined in the driver structure are:

1. Hotplugging callbacks:

- **@probe:** Called to see if the driver is willing to manage a particular interface on a device.
- **@disconnect:** Called when the interface is no longer accessible, usually because its device has been (or is being) disconnected or the driver module is being unloaded.

2. Odd backdoor through usbfs:

- **@ioctl:** Used for drivers that want to talk to userspace through the "usbfs" filesystem. This lets devices provide ways to expose information to user space regardless of where they do (or don't) show up otherwise in the filesystem.

3. Power management (PM) callbacks:

- **@suspend:** Called when the device is going to be suspended.
- **@resume:** Called when the device is being resumed.
- **@reset_resume:** Called when the suspended device has been reset instead of being resumed.

4. Device level operations:

- **@pre_reset:** Called when the device is about to be reset.
- **@post_reset:** Called after the device has been reset

The ioctl interface (2) should be used only if you have a very good reason. Sysfs is preferred these days. The PM callbacks are covered separately in *Power Management for USB*.

Calling conventions

All callbacks are mutually exclusive. There's no need for locking against other USB callbacks. All callbacks are called from a task context. You may sleep. However, it is important that all sleeps have a small fixed upper limit in time. In particular you must not call out to user space and await results.

Hotplugging callbacks

These callbacks are intended to associate and disassociate a driver with an interface. A driver's bond to an interface is exclusive.

The probe() callback

```
int (*probe) (struct usb_interface *intf,
              const struct usb_device_id *id);
```

Accept or decline an interface. If you accept the device return 0, otherwise -ENODEV or -ENXIO. Other error codes should be used only if a genuine error occurred during initialisation which prevented a driver from accepting a device that would else have been accepted. You are strongly encouraged to use usbcore's facility, usb_set_intfdata(), to associate a data structure with an interface, so that you know which internal state and identity you associate with a particular interface. The device will not be suspended and you may do IO to the interface you are called for and endpoint 0 of the device. Device initialisation that doesn't take too long is a good idea here.

The disconnect() callback

```
void (*disconnect) (struct usb_interface *intf);
```

This callback is a signal to break any connection with an interface. You are not allowed any IO to a device after returning from this callback. You also may not do any other operation that may interfere with another driver bound the interface, eg. a power management operation. If you are called due to a physical disconnection, all your URBs will be killed by usbcore. Note that in this case disconnect will be called some time after the physical disconnection. Thus your driver must be prepared to deal with failing IO even prior to the callback.

Device level callbacks

pre_reset

```
int (*pre_reset)(struct usb_interface *intf);
```

A driver or user space is triggering a reset on the device which contains the interface passed as an argument. Cease IO, wait for all outstanding URBs to complete, and save any device state you need to restore. No more URBs may be submitted until the post_reset method is called.

If you need to allocate memory here, use GFP_NOIO or GFP_ATOMIC, if you are in atomic context.

post_reset

```
int (*post_reset)(struct usb_interface *intf);
```

The reset has completed. Restore any saved device state and begin using the device again.

If you need to allocate memory here, use GFP_NOIO or GFP_ATOMIC, if you are in atomic context.

Call sequences

No callbacks other than probe will be invoked for an interface that isn't bound to your driver.

Probe will never be called for an interface bound to a driver. Hence following a successful probe, disconnect will be called before there is another probe for the same interface.

Once your driver is bound to an interface, disconnect can be called at any time except in between pre_reset and post_reset. pre_reset is always followed by post_reset, even if the reset failed or the device has been unplugged.

suspend is always followed by one of: resume, reset_resume, or disconnect.

USB DMA

In Linux 2.5 kernels (and later), USB device drivers have additional control over how DMA may be used to perform I/O operations. The APIs are detailed in the kernel usb programming guide (kerneldoc, from the source code).

API overview

The big picture is that USB drivers can continue to ignore most DMA issues, though they still must provide DMA-ready buffers (see `Documentation/DMA-API-HOWTO.txt`). That's how they've worked through the 2.4 (and earlier) kernels, or they can now be DMA-aware.

DMA-aware usb drivers:

- New calls enable DMA-aware drivers, letting them allocate dma buffers and manage dma mappings for existing dma-ready buffers (see below).
- URBs have an additional "transfer_dma" field, as well as a transfer_flags bit saying if it's valid. (Control requests also have "setup_dma", but drivers must not use it.)
- "usbcore" will map this DMA address, if a DMA-aware driver didn't do it first and set `URB_NO_TRANSFER_DMA_MAP`. HCDs don't manage dma mappings for URBs.
- There's a new "generic DMA API", parts of which are usable by USB device drivers. Never use dma_set_mask() on any USB interface or device; that would potentially break all devices sharing that bus.

Eliminating copies

It's good to avoid making CPUs copy data needlessly. The costs can add up, and effects like cache-trashing can impose subtle penalties.

- If you're doing lots of small data transfers from the same buffer all the time, that can really burn up resources on systems which use an IOMMU to manage the DMA mappings. It can cost MUCH more to set up and tear down the IOMMU mappings with each request than perform the I/O!

  For those specific cases, USB has primitives to allocate less expensive memory. They work like kmalloc and kfree versions that give you the right kind of addresses to store in urb->transfer_buffer and urb->transfer_dma. You'd also set `URB_NO_TRANSFER_DMA_MAP` in urb->transfer_flags:

```
void *usb_alloc_coherent (struct usb_device *dev, size_t size,
        int mem_flags, dma_addr_t *dma);

void usb_free_coherent (struct usb_device *dev, size_t size,
        void *addr, dma_addr_t dma);
```

Most drivers should **NOT** be using these primitives; they don't need to use this type of memory ("dma-coherent"), and memory returned from `kmalloc()` will work just fine.

The memory buffer returned is "dma-coherent"; sometimes you might need to force a consistent memory access ordering by using memory barriers. It's not using a streaming DMA mapping, so it's good for small transfers on systems where the I/O would otherwise thrash an IOMMU mapping. (See `Documentation/DMA-API-HOWTO.txt` for definitions of "coherent" and "streaming" DMA mappings.)

Asking for 1/Nth of a page (as well as asking for N pages) is reasonably space-efficient.

On most systems the memory returned will be uncached, because the semantics of dma-coherent memory require either bypassing CPU caches or using cache hardware with bus-snooping support. While x86 hardware has such bus-snooping, many other systems use software to flush cache lines to prevent DMA conflicts.

- Devices on some EHCI controllers could handle DMA to/from high memory.

  Unfortunately, the current Linux DMA infrastructure doesn't have a sane way to expose these capabilities ... and in any case, HIGHMEM is mostly a design wart specific to x86_32. So your best bet is to ensure you never pass a highmem buffer into a USB driver. That's easy; it's the default behavior. Just don't override it; e.g. with `NETIF_F_HIGHDMA`.

  This may force your callers to do some bounce buffering, copying from high memory to "normal" DMA memory. If you can come up with a good way to fix this issue (for x86_32 machines with over 1 GByte of memory), feel free to submit patches.

Working with existing buffers

Existing buffers aren't usable for DMA without first being mapped into the DMA address space of the device. However, most buffers passed to your driver can safely be used with such DMA mapping. (See the first section of Documentation/DMA-API-HOWTO.txt, titled "What memory is DMA-able?")

- When you're using scatterlists, you can map everything at once. On some systems, this kicks in an IOMMU and turns the scatterlists into single DMA transactions:

```
int usb_buffer_map_sg (struct usb_device *dev, unsigned pipe,
        struct scatterlist *sg, int nents);

void usb_buffer_dmasync_sg (struct usb_device *dev, unsigned pipe,
        struct scatterlist *sg, int n_hw_ents);

void usb_buffer_unmap_sg (struct usb_device *dev, unsigned pipe,
        struct scatterlist *sg, int n_hw_ents);
```

  It's probably easier to use the new `usb_sg_*()` calls, which do the DMA mapping and apply other tweaks to make scatterlist i/o be fast.

- Some drivers may prefer to work with the model that they're mapping large buffers, synchronizing their safe re-use. (If there's no re-use, then let usbcore do the map/unmap.) Large periodic transfers make good examples here, since it's cheaper to just synchronize the buffer than to unmap it each time an urb completes and then re-map it on during resubmission.

  These calls all work with initialized urbs: `urb->dev`, `urb->pipe`, `urb->transfer_buffer`, and `urb->transfer_buffer_length` must all be valid when these calls are used (`urb->setup_packet` must be valid too if urb is a control request):

```
    struct urb *usb_buffer_map (struct urb *urb);

    void usb_buffer_dmasync (struct urb *urb);

    void usb_buffer_unmap (struct urb *urb);
```

The calls manage urb->transfer_dma for you, and set URB_NO_TRANSFER_DMA_MAP so that usbcore won't map or unmap the buffer. They cannot be used for setup_packet buffers in control requests.

Note that several of those interfaces are currently commented out, since they don't have current users. See the source code. Other than the dmasync calls (where the underlying DMA primitives have changed), most of them can easily be commented back in if you want to use them.

USB Request Block (URB)

Revised 2000-Dec-05

Again 2002-Jul-06

Again 2005-Sep-19

Again 2017-Mar-29

Note:

The USB subsystem now has a substantial section at The Linux-USB Host Side API section, generated from the current source code. This particular documentation file isn't complete and may not be updated to the last version; don't rely on it except for a quick overview.

Basic concept or 'What is an URB?'

The basic idea of the new driver is message passing, the message itself is called USB Request Block, or URB for short.

- An URB consists of all relevant information to execute any USB transaction and deliver the data and status back.
- Execution of an URB is inherently an asynchronous operation, i.e. the *usb_submit_urb()* call returns immediately after it has successfully queued the requested action.
- Transfers for one URB can be canceled with *usb_unlink_urb()* at any time.
- Each URB has a completion handler, which is called after the action has been successfully completed or canceled. The URB also contains a context-pointer for passing information to the completion handler.
- Each endpoint for a device logically supports a queue of requests. You can fill that queue, so that the USB hardware can still transfer data to an endpoint while your driver handles completion of another. This maximizes use of USB bandwidth, and supports seamless streaming of data to (or from) devices when using periodic transfer modes.

The URB structure

Some of the fields in struct *urb* are:

```
struct urb
{
// (IN) device and pipe specify the endpoint queue
      struct usb_device *dev;         // pointer to associated USB device
      unsigned int pipe;              // endpoint information

      unsigned int transfer_flags;    // URB_ISO_ASAP, URB_SHORT_NOT_OK, etc.

// (IN) all urbs need completion routines
      void *context;                  // context for completion routine
      usb_complete_t complete;        // pointer to completion routine

// (OUT) status after each completion
      int status;                     // returned status

// (IN) buffer used for data transfers
      void *transfer_buffer;          // associated data buffer
      u32 transfer_buffer_length;     // data buffer length
      int number_of_packets;          // size of iso_frame_desc

// (OUT) sometimes only part of CTRL/BULK/INTR transfer_buffer is used
      u32 actual_length;              // actual data buffer length

// (IN) setup stage for CTRL (pass a struct usb_ctrlrequest)
      unsigned char *setup_packet;    // setup packet (control only)

// Only for PERIODIC transfers (ISO, INTERRUPT)
  // (IN/OUT) start_frame is set unless URB_ISO_ASAP isn't set
      int start_frame;                // start frame
      int interval;                   // polling interval

  // ISO only: packets are only "best effort"; each can have errors
      int error_count;                // number of errors
      struct usb_iso_packet_descriptor iso_frame_desc[0];
};
```

Your driver must create the “pipe” value using values from the appropriate endpoint descriptor in an interface that it’s claimed.

How to get an URB?

URBs are allocated by calling *usb_alloc_urb()*:

```
struct urb *usb_alloc_urb(int isoframes, int mem_flags)
```

Return value is a pointer to the allocated URB, 0 if allocation failed. The parameter isoframes specifies the number of isochronous transfer frames you want to schedule. For CTRL/BULK/INT, use 0. The mem_flags parameter holds standard memory allocation flags, letting you control (among other things) whether the underlying code may block or not.

To free an URB, use *usb_free_urb()*:

```
void usb_free_urb(struct urb *urb)
```

You may free an urb that you’ve submitted, but which hasn’t yet been returned to you in a completion callback. It will automatically be deallocated when it is no longer in use.

What has to be filled in?

Depending on the type of transaction, there are some inline functions defined in `linux/usb.h` to simplify the initialization, such as *usb_fill_control_urb()*, *usb_fill_bulk_urb()* and *usb_fill_int_urb()*.

In general, they need the usb device pointer, the pipe (usual format from usb.h), the transfer buffer, the desired transfer length, the completion handler, and its context. Take a look at the some existing drivers to see how they're used.

Flags:

- For ISO there are two startup behaviors: Specified start_frame or ASAP.
- For ASAP set `URB_ISO_ASAP` in transfer_flags.

If short packets should NOT be tolerated, set `URB_SHORT_NOT_OK` in transfer_flags.

How to submit an URB?

Just call *usb_submit_urb()*:

```
int usb_submit_urb(struct urb *urb, int mem_flags)
```

The `mem_flags` parameter, such as `GFP_ATOMIC`, controls memory allocation, such as whether the lower levels may block when memory is tight.

It immediately returns, either with status 0 (request queued) or some error code, usually caused by the following:

- Out of memory (`-ENOMEM`)
- Unplugged device (`-ENODEV`)
- Stalled endpoint (`-EPIPE`)
- Too many queued ISO transfers (`-EAGAIN`)
- Too many requested ISO frames (`-EFBIG`)
- Invalid INT interval (`-EINVAL`)
- More than one packet for INT (`-EINVAL`)

After submission, `urb->status` is `-EINPROGRESS`; however, you should never look at that value except in your completion callback.

For isochronous endpoints, your completion handlers should (re)submit URBs to the same endpoint with the `URB_ISO_ASAP` flag, using multi-buffering, to get seamless ISO streaming.

How to cancel an already running URB?

There are two ways to cancel an URB you've submitted but which hasn't been returned to your driver yet. For an asynchronous cancel, call *usb_unlink_urb()*:

```
int usb_unlink_urb(struct urb *urb)
```

It removes the urb from the internal list and frees all allocated HW descriptors. The status is changed to reflect unlinking. Note that the URB will not normally have finished when *usb_unlink_urb()* returns; you must still wait for the completion handler to be called.

To cancel an URB synchronously, call *usb_kill_urb()*:

```
void usb_kill_urb(struct urb *urb)
```

It does everything *usb_unlink_urb()* does, and in addition it waits until after the URB has been returned and the completion handler has finished. It also marks the URB as temporarily unusable, so that if the completion handler or anyone else tries to resubmit it they will get a `-EPERM` error. Thus you can be sure that when *usb_kill_urb()* returns, the URB is totally idle.

There is a lifetime issue to consider. An URB may complete at any time, and the completion handler may free the URB. If this happens while *usb_unlink_urb()* or *usb_kill_urb()* is running, it will cause a memory-access violation. The driver is responsible for avoiding this, which often means some sort of lock will be needed to prevent the URB from being deallocated while it is still in use.

On the other hand, since usb_unlink_urb may end up calling the completion handler, the handler must not take any lock that is held when usb_unlink_urb is invoked. The general solution to this problem is to increment the URB's reference count while holding the lock, then drop the lock and call usb_unlink_urb or usb_kill_urb, and then decrement the URB's reference count. You increment the reference count by calling :c:func'usb_get_urb':

```
struct urb *usb_get_urb(struct urb *urb)
```

(ignore the return value; it is the same as the argument) and decrement the reference count by calling *usb_free_urb()*. Of course, none of this is necessary if there's no danger of the URB being freed by the completion handler.

What about the completion handler?

The handler is of the following type:

```
typedef void (*usb_complete_t)(struct urb *)
```

I.e., it gets the URB that caused the completion call. In the completion handler, you should have a look at `urb->status` to detect any USB errors. Since the context parameter is included in the URB, you can pass information to the completion handler.

Note that even when an error (or unlink) is reported, data may have been transferred. That's because USB transfers are packetized; it might take sixteen packets to transfer your 1KByte buffer, and ten of them might have transferred successfully before the completion was called.

Warning: *NEVER SLEEP IN A COMPLETION HANDLER.*
These are often called in atomic context.

In the current kernel, completion handlers run with local interrupts disabled, but in the future this will be changed, so don't assume that local IRQs are always disabled inside completion handlers.

How to do isochronous (ISO) transfers?

Besides the fields present on a bulk transfer, for ISO, you also also have to set `urb->interval` to say how often to make transfers; it's often one per frame (which is once every microframe for highspeed devices). The actual interval used will be a power of two that's no bigger than what you specify. You can use the *usb_fill_int_urb()* macro to fill most ISO transfer fields.

For ISO transfers you also have to fill a `usb_iso_packet_descriptor` structure, allocated at the end of the URB by *usb_alloc_urb()*, for each packet you want to schedule.

The *usb_submit_urb()* call modifies `urb->interval` to the implemented interval value that is less than or equal to the requested interval value. If `URB_ISO_ASAP` scheduling is used, `urb->start_frame` is also updated.

For each entry you have to specify the data offset for this frame (base is transfer_buffer), and the length you want to write/expect to read. After completion, actual_length contains the actual transferred length and status contains the resulting status for the ISO transfer for this frame. It is allowed to specify a varying length from frame to frame (e.g. for audio synchronisation/adaptive transfer rates). You can also use the length 0 to omit one or more frames (striping).

For scheduling you can choose your own start frame or `URB_ISO_ASAP`. As explained earlier, if you always keep at least one URB queued and your completion keeps (re)submitting a later URB, you'll get smooth ISO streaming (if usb bandwidth utilization allows).

If you specify your own start frame, make sure it's several frames in advance of the current frame. You might want this model if you're synchronizing ISO data with some other event stream.

How to start interrupt (INT) transfers?

Interrupt transfers, like isochronous transfers, are periodic, and happen in intervals that are powers of two (1, 2, 4 etc) units. Units are frames for full and low speed devices, and microframes for high speed ones. You can use the *`usb_fill_int_urb()`* macro to fill INT transfer fields.

The *`usb_submit_urb()`* call modifies `urb->interval` to the implemented interval value that is less than or equal to the requested interval value.

In Linux 2.6, unlike earlier versions, interrupt URBs are not automagically restarted when they complete. They end when the completion handler is called, just like other URBs. If you want an interrupt URB to be restarted, your completion handler must resubmit it. s

Power Management for USB

Author Alan Stern <stern@rowland.harvard.edu>

Date Last-updated: February 2014

What is Power Management?

Power Management (PM) is the practice of saving energy by suspending parts of a computer system when they aren't being used. While a component is `suspended` it is in a nonfunctional low-power state; it might even be turned off completely. A suspended component can be `resumed` (returned to a functional full-power state) when the kernel needs to use it. (There also are forms of PM in which components are placed in a less functional but still usable state instead of being suspended; an example would be reducing the CPU's clock rate. This document will not discuss those other forms.)

When the parts being suspended include the CPU and most of the rest of the system, we speak of it as a "system suspend". When a particular device is turned off while the system as a whole remains running, we call it a "dynamic suspend" (also known as a "runtime suspend" or "selective suspend"). This document concentrates mostly on how dynamic PM is implemented in the USB subsystem, although system PM is covered to some extent (see `Documentation/power/*.txt` for more information about system PM).

System PM support is present only if the kernel was built with `CONFIG_SUSPEND` or `CONFIG_HIBERNATION` enabled. Dynamic PM support

for USB is present whenever the kernel was built with `CONFIG_PM` enabled.

[Historically, dynamic PM support for USB was present only if the kernel had been built with `CONFIG_USB_SUSPEND` enabled (which depended on `CONFIG_PM_RUNTIME`). Starting with the 3.10 kernel release, dynamic PM support for USB was present whenever the kernel was built with `CONFIG_PM_RUNTIME` enabled. The `CONFIG_USB_SUSPEND` option had been eliminated.]

What is Remote Wakeup?

When a device has been suspended, it generally doesn't resume until the computer tells it to. Likewise, if the entire computer has been suspended, it generally doesn't resume until the user tells it to, say by pressing a power button or opening the cover.

However some devices have the capability of resuming by themselves, or asking the kernel to resume them, or even telling the entire computer to resume. This capability goes by several names such as "Wake On LAN"; we will refer to it generically as "remote wakeup". When a device is enabled for remote wakeup and it is suspended, it may resume itself (or send a request to be resumed) in response to some external event. Examples include a suspended keyboard resuming when a key is pressed, or a suspended USB hub resuming when a device is plugged in.

When is a USB device idle?

A device is idle whenever the kernel thinks it's not busy doing anything important and thus is a candidate for being suspended. The exact definition depends on the device's driver; drivers are allowed to declare that a device isn't idle even when there's no actual communication taking place. (For example, a hub isn't considered idle unless all the devices plugged into that hub are already suspended.) In addition, a device isn't considered idle so long as a program keeps its usbfs file open, whether or not any I/O is going on.

If a USB device has no driver, its usbfs file isn't open, and it isn't being accessed through sysfs, then it definitely is idle.

Forms of dynamic PM

Dynamic suspends occur when the kernel decides to suspend an idle device. This is called `autosuspend` for short. In general, a device won't be autosuspended unless it has been idle for some minimum period of time, the so-called idle-delay time.

Of course, nothing the kernel does on its own initiative should prevent the computer or its devices from working properly. If a device has been autosuspended and a program tries to use it, the kernel will automatically resume the device (autoresume). For the same reason, an autosuspended device will usually have remote wakeup enabled, if the device supports remote wakeup.

It is worth mentioning that many USB drivers don't support autosuspend. In fact, at the time of this writing (Linux 2.6.23) the only drivers which do support it are the hub driver, kaweth, asix, usblp, usblcd, and usb-skeleton (which doesn't count). If a non-supporting driver is bound to a device, the device won't be autosuspended. In effect, the kernel pretends the device is never idle.

We can categorize power management events in two broad classes: external and internal. External events are those triggered by some agent outside the USB stack: system suspend/resume (triggered by userspace), manual dynamic resume (also triggered by userspace), and remote wakeup (triggered by the device). Internal events are those triggered within the USB stack: autosuspend and autoresume. Note that all dynamic suspend events are internal; external agents are not allowed to issue dynamic suspends.

The user interface for dynamic PM

The user interface for controlling dynamic PM is located in the `power/` subdirectory of each USB device's sysfs directory, that is, in `/sys/bus/usb/devices/.../power/` where "…" is the device's ID. The relevant attribute files are: wakeup, control, and `autosuspend_delay_ms`. (There may also be a file named `level`; this file was deprecated as of the 2.6.35 kernel and replaced by the `control` file. In 2.6.38 the `autosuspend` file will be deprecated and replaced by the `autosuspend_delay_ms` file. The only difference is that the newer file expresses the delay in milliseconds whereas the older file uses seconds. Confusingly, both files are present in 2.6.37 but only `autosuspend` works.)

`power/wakeup`

> This file is empty if the device does not support remote wakeup. Otherwise the file contains either the word `enabled` or the word `disabled`, and you can write those words to the file. The setting determines whether or not remote wakeup will be enabled when the device is next suspended. (If the setting is changed while the device is suspended, the change won't take effect until the following suspend.)

`power/control`

> This file contains one of two words: `on` or `auto`. You can write those words to the file to change the device's setting.

- on means that the device should be resumed and autosuspend is not allowed. (Of course, system suspends are still allowed.)
- auto is the normal state in which the kernel is allowed to autosuspend and autoresume the device.

(In kernels up to 2.6.32, you could also specify suspend, meaning that the device should remain suspended and autoresume was not allowed. This setting is no longer supported.)

power/autosuspend_delay_ms

This file contains an integer value, which is the number of milliseconds the device should remain idle before the kernel will autosuspend it (the idle-delay time). The default is 2000. 0 means to autosuspend as soon as the device becomes idle, and negative values mean never to autosuspend. You can write a number to the file to change the autosuspend idle-delay time.

Writing -1 to power/autosuspend_delay_ms and writing on to power/control do essentially the same thing – they both prevent the device from being autosuspended. Yes, this is a redundancy in the API.

(In 2.6.21 writing 0 to power/autosuspend would prevent the device from being autosuspended; the behavior was changed in 2.6.22. The power/autosuspend attribute did not exist prior to 2.6.21, and the power/level attribute did not exist prior to 2.6.22. power/control was added in 2.6.34, and power/autosuspend_delay_ms was added in 2.6.37 but did not become functional until 2.6.38.)

Changing the default idle-delay time

The default autosuspend idle-delay time (in seconds) is controlled by a module parameter in usbcore. You can specify the value when usbcore is loaded. For example, to set it to 5 seconds instead of 2 you would do:

```
modprobe usbcore autosuspend=5
```

Equivalently, you could add to a configuration file in /etc/modprobe.d a line saying:

```
options usbcore autosuspend=5
```

Some distributions load the usbcore module very early during the boot process, by means of a program or script running from an initramfs image. To alter the parameter value you would have to rebuild that image.

If usbcore is compiled into the kernel rather than built as a loadable module, you can add:

```
usbcore.autosuspend=5
```

to the kernel's boot command line.

Finally, the parameter value can be changed while the system is running. If you do:

```
echo 5 >/sys/module/usbcore/parameters/autosuspend
```

then each new USB device will have its autosuspend idle-delay initialized to 5. (The idle-delay values for already existing devices will not be affected.)

Setting the initial default idle-delay to -1 will prevent any autosuspend of any USB device. This has the benefit of allowing you then to enable autosuspend for selected devices.

Warnings

The USB specification states that all USB devices must support power management. Nevertheless, the sad fact is that many devices do not support it very well. You can suspend them all right, but when you try to resume them they disconnect themselves from the USB bus or they stop working entirely. This seems to be especially prevalent among printers and scanners, but plenty of other types of device have the same deficiency.

For this reason, by default the kernel disables autosuspend (the `power/control` attribute is initialized to on) for all devices other than hubs. Hubs, at least, appear to be reasonably well-behaved in this regard.

(In 2.6.21 and 2.6.22 this wasn't the case. Autosuspend was enabled by default for almost all USB devices. A number of people experienced problems as a result.)

This means that non-hub devices won't be autosuspended unless the user or a program explicitly enables it. As of this writing there aren't any widespread programs which will do this; we hope that in the near future device managers such as HAL will take on this added responsibility. In the meantime you can always carry out the necessary operations by hand or add them to a udev script. You can also change the idle-delay time; 2 seconds is not the best choice for every device.

If a driver knows that its device has proper suspend/resume support, it can enable autosuspend all by itself. For example, the video driver for a laptop's webcam might do this (in recent kernels they do), since these devices are rarely used and so should normally be autosuspended.

Sometimes it turns out that even when a device does work okay with autosuspend there are still problems. For example, the usbhid driver, which manages keyboards and mice, has autosuspend support. Tests with a number of keyboards show that typing on a suspended keyboard, while causing the keyboard to do a remote wakeup all right, will nonetheless frequently result in lost keystrokes. Tests with mice show that some of them will issue a remote-wakeup request in response to button presses but not to motion, and some in response to neither.

The kernel will not prevent you from enabling autosuspend on devices that can't handle it. It is even possible in theory to damage a device by suspending it at the wrong time. (Highly unlikely, but possible.) Take care.

The driver interface for Power Management

The requirements for a USB driver to support external power management are pretty modest; the driver need only define:

```
.suspend
.resume
.reset_resume
```

methods in its *`usb_driver`* structure, and the `reset_resume` method is optional. The methods' jobs are quite simple:

- The `suspend` method is called to warn the driver that the device is going to be suspended. If the driver returns a negative error code, the suspend will be aborted. Normally the driver will return 0, in which case it must cancel all outstanding URBs (*`usb_kill_urb()`*) and not submit any more.
- The `resume` method is called to tell the driver that the device has been resumed and the driver can return to normal operation. URBs may once more be submitted.
- The `reset_resume` method is called to tell the driver that the device has been resumed and it also has been reset. The driver should redo any necessary device initialization, since the device has probably lost most or all of its state (although the interfaces will be in the same altsettings as before the suspend).

If the device is disconnected or powered down while it is suspended, the `disconnect` method will be called instead of the `resume` or `reset_resume` method. This is also quite likely to happen when waking up from hibernation, as many systems do not maintain suspend current to the USB host controllers during hibernation. (It's possible to work around the hibernation-forces-disconnect problem by using the USB Persist facility.)

The `reset_resume` method is used by the USB Persist facility (see *USB device persistence during system suspend*) and it can also be used under certain circumstances when `CONFIG_USB_PERSIST` is not enabled. Currently, if a device is reset during a resume and the driver does not have a `reset_resume` method, the driver won't receive any notification about the resume. Later kernels will call the driver's `disconnect` method; 2.6.23 doesn't do this.

USB drivers are bound to interfaces, so their `suspend` and `resume` methods get called when the interfaces are suspended or resumed. In principle one might want to suspend some interfaces on a device (i.e., force the drivers for those interface to stop all activity) without suspending the other interfaces. The USB core doesn't allow this; all interfaces are suspended when the device itself is suspended and all interfaces are resumed when the device is resumed. It isn't possible to suspend or resume some but not all of a device's interfaces. The closest you can come is to unbind the interfaces' drivers.

The driver interface for autosuspend and autoresume

To support autosuspend and autoresume, a driver should implement all three of the methods listed above. In addition, a driver indicates that it supports autosuspend by setting the `.supports_autosuspend` flag in its usb_driver structure. It is then responsible for informing the USB core whenever one of its interfaces becomes busy or idle. The driver does so by calling these six functions:

```
int  usb_autopm_get_interface(struct usb_interface *intf);
void usb_autopm_put_interface(struct usb_interface *intf);
int  usb_autopm_get_interface_async(struct usb_interface *intf);
void usb_autopm_put_interface_async(struct usb_interface *intf);
void usb_autopm_get_interface_no_resume(struct usb_interface *intf);
void usb_autopm_put_interface_no_suspend(struct usb_interface *intf);
```

The functions work by maintaining a usage counter in the usb_interface's embedded device structure. When the counter is > 0 then the interface is deemed to be busy, and the kernel will not autosuspend the interface's device. When the usage counter is = 0 then the interface is considered to be idle, and the kernel may autosuspend the device.

Drivers need not be concerned about balancing changes to the usage counter; the USB core will undo any remaining "get"s when a driver is unbound from its interface. As a corollary, drivers must not call any of the `usb_autopm_*` functions after their `disconnect` routine has returned.

Drivers using the async routines are responsible for their own synchronization and mutual exclusion.

> *`usb_autopm_get_interface()`* increments the usage counter and does an autoresume if the device is suspended. If the autoresume fails, the counter is decremented back.
>
> *`usb_autopm_put_interface()`* decrements the usage counter and attempts an autosuspend if the new value is = 0.
>
> *`usb_autopm_get_interface_async()`* and *`usb_autopm_put_interface_async()`* do almost the same things as their non-async counterparts. The big difference is that they use a workqueue to do the resume or suspend part of their jobs. As a result they can be called in an atomic context, such as an URB's completion handler, but when they return the device will generally not yet be in the desired state.
>
> *`usb_autopm_get_interface_no_resume()`* and *`usb_autopm_put_interface_no_suspend()`* merely increment or decrement the usage counter; they do not attempt to carry out an autoresume or an autosuspend. Hence they can be called in an atomic context.

The simplest usage pattern is that a driver calls *`usb_autopm_get_interface()`* in its open routine and *`usb_autopm_put_interface()`* in its close or release routine. But other patterns are possible.

The autosuspend attempts mentioned above will often fail for one reason or another. For example, the `power/control` attribute might be set to on, or another interface in the same device might not be idle. This is perfectly normal. If the reason for failure was that the device hasn't been idle for long enough, a timer is scheduled to carry out the operation automatically when the autosuspend idle-delay has expired.

Autoresume attempts also can fail, although failure would mean that the device is no longer present or operating properly. Unlike autosuspend, there's no idle-delay for an autoresume.

Other parts of the driver interface

Drivers can enable autosuspend for their devices by calling:

```
usb_enable_autosuspend(struct usb_device *udev);
```

in their `probe()` routine, if they know that the device is capable of suspending and resuming correctly. This is exactly equivalent to writing `auto` to the device's `power/control` attribute. Likewise, drivers can disable autosuspend by calling:

```
usb_disable_autosuspend(struct usb_device *udev);
```

This is exactly the same as writing `on` to the `power/control` attribute.

Sometimes a driver needs to make sure that remote wakeup is enabled during autosuspend. For example, there's not much point autosuspending a keyboard if the user can't cause the keyboard to do a remote wakeup by typing on it. If the driver sets `intf->needs_remote_wakeup` to 1, the kernel won't autosuspend the device if remote wakeup isn't available. (If the device is already autosuspended, though, setting this flag won't cause the kernel to autoresume it. Normally a driver would set this flag in its `probe` method, at which time the device is guaranteed not to be autosuspended.)

If a driver does its I/O asynchronously in interrupt context, it should call *`usb_autopm_get_interface_async()`* before starting output and *`usb_autopm_put_interface_async()`* when the output queue drains. When it receives an input event, it should call:

```
usb_mark_last_busy(struct usb_device *udev);
```

in the event handler. This tells the PM core that the device was just busy and therefore the next autosuspend idle-delay expiration should be pushed back. Many of the usb_autopm_* routines also make this call, so drivers need to worry only when interrupt-driven input arrives.

Asynchronous operation is always subject to races. For example, a driver may call the *`usb_autopm_get_interface_async()`* routine at a time when the core has just finished deciding the device has been idle for long enough but not yet gotten around to calling the driver's `suspend` method. The `suspend` method must be responsible for synchronizing with the I/O request routine and the URB completion handler; it should cause autosuspends to fail with -EBUSY if the driver needs to use the device.

External suspend calls should never be allowed to fail in this way, only autosuspend calls. The driver can tell them apart by applying the `PMSG_IS_AUTO()` macro to the message argument to the `suspend` method; it will return True for internal PM events (autosuspend) and False for external PM events.

Mutual exclusion

For external events – but not necessarily for autosuspend or autoresume – the device semaphore (udev->dev.sem) will be held when a `suspend` or `resume` method is called. This implies that external suspend/resume events are mutually exclusive with calls to `probe`, `disconnect`, `pre_reset`, and `post_reset`; the USB core guarantees that this is true of autosuspend/autoresume events as well.

If a driver wants to block all suspend/resume calls during some critical section, the best way is to lock the device and call *`usb_autopm_get_interface()`* (and do the reverse at the end of the critical section). Holding the device semaphore will block all external PM calls, and the *`usb_autopm_get_interface()`* will prevent any internal PM calls, even if it fails. (Exercise: Why?)

Interaction between dynamic PM and system PM

Dynamic power management and system power management can interact in a couple of ways.

Firstly, a device may already be autosuspended when a system suspend occurs. Since system suspends are supposed to be as transparent as possible, the device should remain suspended following the system resume. But this theory may not work out well in practice; over time the kernel's behavior in this regard has changed. As of 2.6.37 the policy is to resume all devices during a system resume and let them handle their own runtime suspends afterward.

Secondly, a dynamic power-management event may occur as a system suspend is underway. The window for this is short, since system suspends don't take long (a few seconds usually), but it can happen. For example, a suspended device may send a remote-wakeup signal while the system is suspending. The remote wakeup may succeed, which would cause the system suspend to abort. If the remote wakeup doesn't succeed, it may still remain active and thus cause the system to resume as soon as the system suspend is complete. Or the remote wakeup may fail and get lost. Which outcome occurs depends on timing and on the hardware and firmware design.

xHCI hardware link PM

xHCI host controller provides hardware link power management to usb2.0 (xHCI 1.0 feature) and usb3.0 devices which support link PM. By enabling hardware LPM, the host can automatically put the device into lower power state(L1 for usb2.0 devices, or U1/U2 for usb3.0 devices), which state device can enter and resume very quickly.

The user interface for controlling hardware LPM is located in the `power/` subdirectory of each USB device's sysfs directory, that is, in `/sys/bus/usb/devices/.../power/` where "…" is the device's ID. The relevant attribute files are `usb2_hardware_lpm` and `usb3_hardware_lpm`.

`power/usb2_hardware_lpm`

> When a USB2 device which support LPM is plugged to a xHCI host root hub which support software LPM, the host will run a software LPM test for it; if the device enters L1 state and resume successfully and the host supports USB2 hardware LPM, this file will show up and driver will enable hardware LPM for the device. You can write y/Y/1 or n/N/0 to the file to enable/disable USB2 hardware LPM manually. This is for test purpose mainly.

`power/usb3_hardware_lpm_u1` `power/usb3_hardware_lpm_u2`

> When a USB 3.0 lpm-capable device is plugged in to a xHCI host which supports link PM, it will check if U1 and U2 exit latencies have been set in the BOS descriptor; if the check is passed and the host supports USB3 hardware LPM, USB3 hardware LPM will be enabled for the device and these files will be created. The files hold a string value (enable or disable) indicating whether or not USB3 hardware LPM U1 or U2 is enabled for the device.

USB Port Power Control

In addition to suspending endpoint devices and enabling hardware controlled link power management, the USB subsystem also has the capability to disable power to ports under some conditions. Power is controlled through `Set/ClearPortFeature(PORT_POWER)` requests to a hub. In the case of a root or platform-internal hub the host controller driver translates `PORT_POWER` requests into platform firmware (ACPI) method calls to set the port power state. For more background see the Linux Plumbers Conference 2012 slides [1] and video [2]:

Upon receiving a `ClearPortFeature(PORT_POWER)` request a USB port is logically off, and may trigger the actual loss of VBUS to the port [3]. VBUS may be maintained in the case where a hub gangs multiple ports into a shared power well causing power to remain until all ports in the gang are turned off. VBUS may also be maintained by hub ports configured for a charging application. In any event a logically off port will lose connection with its device, not respond to hotplug events, and not respond to remote wakeup events.

[1] http://dl.dropbox.com/u/96820575/sarah-sharp-lpt-port-power-off2-mini.pdf
[2] http://linuxplumbers.ubicast.tv/videos/usb-port-power-off-kerneluserspace-api/
[3] USB 3.1 Section 10.12
wakeup note: if a device is configured to send wakeup events the port power control implementation will block poweroff attempts on that port.

Warning: *turning off a port may result in the inability to hot add a device. Please see "User Interface fo Power Control" for details.*

As far as the effect on the device itself it is similar to what a device goes through during system suspend, i.e. the power session is lost. Any USB device or driver that misbehaves with system suspend will be similarly affected by a port power cycle event. For this reason the implementation shares the same device recovery path (and honors the same quirks) as the system resume path for the hub.

User Interface for Port Power Control

The port power control mechanism uses the PM runtime system. Poweroff is requested by clearing the `power/pm_qos_no_power_off` flag of the port device (defaults to 1). If the port is disconnected it will immediately receive a `ClearPortFeature(PORT_POWER)` request. Otherwise, it will honor the pm runtime rules and require the attached child device and all descendants to be suspended. This mechanism is dependent on the hub advertising port power switching in its hub descriptor (wHubCharacteristics logical power switching mode field).

Note, some interface devices/drivers do not support autosuspend. Userspace may need to unbind the interface drivers before the *usb_device* will suspend. An unbound interface device is suspended by default. When unbinding, be careful to unbind interface drivers, not the driver of the parent usb device. Also, leave hub interface drivers bound. If the driver for the usb device (not interface) is unbound the kernel is no longer able to resume the device. If a hub interface driver is unbound, control of its child ports is lost and all attached child-devices will disconnect. A good rule of thumb is that if the 'driver/module' link for a device points to `/sys/module/usbcore` then unbinding it will interfere with port power control.

Example of the relevant files for port power control. Note, in this example these files are relative to a usb hub device (prefix):

```
prefix=/sys/devices/pci0000:00/0000:00:14.0/usb3/3-1

                 attached child device +
             hub port device +         |
hub interface device +       |         |
                     v       v         v
             $prefix/3-1:1.0/3-1-port1/device

$prefix/3-1:1.0/3-1-port1/power/pm_qos_no_power_off
$prefix/3-1:1.0/3-1-port1/device/power/control
$prefix/3-1:1.0/3-1-port1/device/3-1.1:<intf0>/driver/unbind
$prefix/3-1:1.0/3-1-port1/device/3-1.1:<intf1>/driver/unbind
...
$prefix/3-1:1.0/3-1-port1/device/3-1.1:<intfN>/driver/unbind
```

In addition to these files some ports may have a 'peer' link to a port on another hub. The expectation is that all superspeed ports have a hi-speed peer:

```
$prefix/3-1:1.0/3-1-port1/peer -> ../../../../usb2/2-1/2-1:1.0/2-1-port1
../../../../usb2/2-1/2-1:1.0/2-1-port1/peer -> ../../../../usb3/3-1/3-1:1.0/3-1-port1
```

Distinct from 'companion ports', or 'ehci/xhci shared switchover ports' peer ports are simply the hi-speed and superspeed interface pins that are combined into a single usb3 connector. Peer ports share the same ancestor XHCI device.

While a superspeed port is powered off a device may downgrade its connection and attempt to connect to the hi-speed pins. The implementation takes steps to prevent this:

1. Port suspend is sequenced to guarantee that hi-speed ports are powered-off before their superspeed peer is permitted to power-off. The implication is that the setting `pm_qos_no_power_off` to zero on a superspeed port may not cause the port to power-off until its highspeed peer has gone to its runtime

suspend state. Userspace must take care to order the suspensions if it wants to guarantee that a superspeed port will power-off.

2. Port resume is sequenced to force a superspeed port to power-on prior to its highspeed peer.
3. Port resume always triggers an attached child device to resume. After a power session is lost the device may have been removed, or need reset. Resuming the child device when the parent port regains power resolves those states and clamps the maximum port power cycle frequency at the rate the child device can suspend (autosuspend-delay) and resume (reset-resume latency).

Sysfs files relevant for port power control:

`<hubdev-portX>/power/pm_qos_no_power_off`: This writable flag controls the state of an idle port. Once all children and descendants have suspended the port may suspend/poweroff provided that pm_qos_no_power_off is '0'. If pm_qos_no_power_off is '1' the port will remain active/powered regardless of the stats of descendants. Defaults to 1.

`<hubdev-portX>/power/runtime_status`: This file reflects whether the port is 'active' (power is on) or 'suspended' (logically off). There is no indication to userspace whether VBUS is still supplied.

`<hubdev-portX>/connect_type`: An advisory read-only flag to userspace indicating the location and connection type of the port. It returns one of four values 'hotplug', 'hardwired', 'not used', and 'unknown'. All values, besides unknown, are set by platform firmware.

`hotplug` indicates an externally connectable/visible port on the platform. Typically userspace would choose to keep such a port powered to handle new device connection events.

`hardwired` refers to a port that is not visible but connectable. Examples are internal ports for USB bluetooth that can be disconnected via an external switch or a port with a hardwired USB camera. It is expected to be safe to allow these ports to suspend provided pm_qos_no_power_off is coordinated with any switch that gates connections. Userspace must arrange for the device to be connected prior to the port powering off, or to activate the port prior to enabling connection via a switch.

`not used` refers to an internal port that is expected to never have a device connected to it. These may be empty internal ports, or ports that are not physically exposed on a platform. Considered safe to be powered-off at all times.

`unknown` means platform firmware does not provide information for this port. Most commonly refers to external hub ports which should be considered 'hotplug' for policy decisions.

Note:

- *since we are relying on the BIOS to get this ACPI information correct, the USB port descriptions may be missing or wrong.*
- *Take care in clearing `pm_qos_no_power_off`. Once power is off this port will not respond to new connect events.*

Once a child device is attached additional constraints are applied before the port is allowed to poweroff.

`<child>/power/control`: Must be `auto`, and the port will not power down until `<child>/power/runtime_status` reflects the 'suspended' state. Default value is controlled by child device driver.

`<child>/power/persist`: This defaults to 1 for most devices and indicates if kernel can persist the device's configuration across a power session loss (suspend / port-power event). When this value is `0` (quirky devices), port poweroff is disabled.

`<child>/driver/unbind`: Wakeup capable devices will block port poweroff. At this time the only mechanism to clear the usb-internal wakeup-capability for an interface device is to unbind its driver.

Summary of poweroff pre-requisite settings relative to a port device:

```
echo 0 > power/pm_qos_no_power_off
echo 0 > peer/power/pm_qos_no_power_off # if it exists
echo auto > power/control # this is the default value
echo auto > <child>/power/control
echo 1 > <child>/power/persist # this is the default value
```

Suggested Userspace Port Power Policy

As noted above userspace needs to be careful and deliberate about what ports are enabled for poweroff.

The default configuration is that all ports start with `power/pm_qos_no_power_off` set to 1 causing ports to always remain active.

Given confidence in the platform firmware's description of the ports (ACPI _PLD record for a port populates 'connect_type') userspace can clear pm_qos_no_power_off for all 'not used' ports. The same can be done for 'hardwired' ports provided poweroff is coordinated with any connection switch for the port.

A more aggressive userspace policy is to enable USB port power off for all ports (set `<hubdev-portX>/power/pm_qos_no_power_off` to `0`) when some external factor indicates the user has stopped interacting with the system. For example, a distro may want to enable power off all USB ports when the screen blanks, and re-power them when the screen becomes active. Smart phones and tablets may want to power off USB ports when the user pushes the power button.

USB hotplugging

Linux Hotplugging

In hotpluggable busses like USB (and Cardbus PCI), end-users plug devices into the bus with power on. In most cases, users expect the devices to become immediately usable. That means the system must do many things, including:

- Find a driver that can handle the device. That may involve loading a kernel module; newer drivers can use module-init-tools to publish their device (and class) support to user utilities.
- Bind a driver to that device. Bus frameworks do that using a device driver's probe() routine.
- Tell other subsystems to configure the new device. Print queues may need to be enabled, networks brought up, disk partitions mounted, and so on. In some cases these will be driver-specific actions.

This involves a mix of kernel mode and user mode actions. Making devices be immediately usable means that any user mode actions can't wait for an administrator to do them: the kernel must trigger them, either passively (triggering some monitoring daemon to invoke a helper program) or actively (calling such a user mode helper program directly).

Those triggered actions must support a system's administrative policies; such programs are called "policy agents" here. Typically they involve shell scripts that dispatch to more familiar administration tools.

Because some of those actions rely on information about drivers (metadata) that is currently available only when the drivers are dynamically linked, you get the best hotplugging when you configure a highly modular system.

Kernel Hotplug Helper (`/sbin/hotplug`)

There is a kernel parameter: `/proc/sys/kernel/hotplug`, which normally holds the pathname `/sbin/hotplug`. That parameter names a program which the kernel may invoke at various times.

The /sbin/hotplug program can be invoked by any subsystem as part of its reaction to a configuration change, from a thread in that subsystem. Only one parameter is required: the name of a subsystem being notified of some kernel event. That name is used as the first key for further event dispatch; any other argument and environment parameters are specified by the subsystem making that invocation.

Hotplug software and other resources is available at:

> http://linux-hotplug.sourceforge.net

Mailing list information is also available at that site.

USB Policy Agent

The USB subsystem currently invokes `/sbin/hotplug` when USB devices are added or removed from system. The invocation is done by the kernel hub workqueue [hub_wq], or else as part of root hub initialization (done by init, modprobe, kapmd, etc). Its single command line parameter is the string "usb", and it passes these environment variables:

ACTION	add, remove
PRODUCT	USB vendor, product, and version codes (hex)
TYPE	device class codes (decimal)
INTERFACE	interface 0 class codes (decimal)

If "usbdevfs" is configured, DEVICE and DEVFS are also passed. DEVICE is the pathname of the device, and is useful for devices with multiple and/or alternate interfaces that complicate driver selection. By design, USB hotplugging is independent of `usbdevfs`: you can do most essential parts of USB device setup without using that filesystem, and without running a user mode daemon to detect changes in system configuration.

Currently available policy agent implementations can load drivers for modules, and can invoke driver-specific setup scripts. The newest ones leverage USB module-init-tools support. Later agents might unload drivers.

USB Modutils Support

Current versions of module-init-tools will create a `modules.usbmap` file which contains the entries from each driver's `MODULE_DEVICE_TABLE`. Such files can be used by various user mode policy agents to make sure all the right driver modules get loaded, either at boot time or later.

See `linux/usb.h` for full information about such table entries; or look at existing drivers. Each table entry describes one or more criteria to be used when matching a driver to a device or class of devices. The specific criteria are identified by bits set in "match_flags", paired with field values. You can construct the criteria directly, or with macros such as these, and use driver_info to store more information:

```
USB_DEVICE (vendorId, productId)
    ... matching devices with specified vendor and product ids
USB_DEVICE_VER (vendorId, productId, lo, hi)
    ... like USB_DEVICE with lo <= productversion <= hi
USB_INTERFACE_INFO (class, subclass, protocol)
    ... matching specified interface class info
USB_DEVICE_INFO (class, subclass, protocol)
    ... matching specified device class info
```

A short example, for a driver that supports several specific USB devices and their quirks, might have a MODULE_DEVICE_TABLE like this:

```
static const struct usb_device_id mydriver_id_table[] = {
    { USB_DEVICE (0x9999, 0xaaaa), driver_info: QUIRK_X },
    { USB_DEVICE (0xbbbb, 0x8888), driver_info: QUIRK_Y|QUIRK_Z },
    ...
    { } /* end with an all-zeroes entry */
};
MODULE_DEVICE_TABLE(usb, mydriver_id_table);
```

Most USB device drivers should pass these tables to the USB subsystem as well as to the module management subsystem. Not all, though: some driver frameworks connect using interfaces layered over USB, and so they won't need such a struct *usb_driver*.

Drivers that connect directly to the USB subsystem should be declared something like this:

```
static struct usb_driver mydriver = {
    .name           = "mydriver",
    .id_table       = mydriver_id_table,
    .probe          = my_probe,
    .disconnect     = my_disconnect,

    /*
    if using the usb chardev framework:
        .minor              = MY_USB_MINOR_START,
        .fops               = my_file_ops,
    if exposing any operations through usbdevfs:
        .ioctl              = my_ioctl,
    */
};
```

When the USB subsystem knows about a driver's device ID table, it's used when choosing drivers to probe(). The thread doing new device processing checks drivers' device ID entries from the `MODULE_DEVICE_TABLE` against interface and device descriptors for the device. It will only call `probe()` if there is a match, and the third argument to `probe()` will be the entry that matched.

If you don't provide an `id_table` for your driver, then your driver may get probed for each new device; the third parameter to `probe()` will be `NULL`.

USB device persistence during system suspend

Author Alan Stern <stern@rowland.harvard.edu>

Date September 2, 2006 (Updated February 25, 2008)

What is the problem?

According to the USB specification, when a USB bus is suspended the bus must continue to supply suspend current (around 1-5 mA). This is so that devices can maintain their internal state and hubs can detect connect-change events (devices being plugged in or unplugged). The technical term is "power session".

If a USB device's power session is interrupted then the system is required to behave as though the device has been unplugged. It's a conservative approach; in the absence of suspend current the computer has no way to know what has actually happened. Perhaps the same device is still attached or perhaps it was removed and a different device plugged into the port. The system must assume the worst.

By default, Linux behaves according to the spec. If a USB host controller loses power during a system suspend, then when the system wakes up all the devices attached to that controller are treated as though they had disconnected. This is always safe and it is the "officially correct" thing to do.

For many sorts of devices this behavior doesn't matter in the least. If the kernel wants to believe that your USB keyboard was unplugged while the system was asleep and a new keyboard was plugged in when the system woke up, who cares? It'll still work the same when you type on it.

Unfortunately problems _can_ arise, particularly with mass-storage devices. The effect is exactly the same as if the device really had been unplugged while the system was suspended. If you had a mounted filesystem on the device, you're out of luck – everything in that filesystem is now inaccessible. This is especially annoying if your root filesystem was located on the device, since your system will instantly crash.

Loss of power isn't the only mechanism to worry about. Anything that interrupts a power session will have the same effect. For example, even though suspend current may have been maintained while the system was asleep, on many systems during the initial stages of wakeup the firmware (i.e., the BIOS) resets the motherboard's USB host controllers. Result: all the power sessions are destroyed and again it's as though you had unplugged all the USB devices. Yes, it's entirely the BIOS's fault, but that doesn't do _you_ any good unless you can convince the BIOS supplier to fix the problem (lots of luck!).

On many systems the USB host controllers will get reset after a suspend-to-RAM. On almost all systems, no suspend current is available during hibernation (also known as swsusp or suspend-to-disk). You can check the kernel log after resuming to see if either of these has happened; look for lines saying "root hub lost power or was reset".

In practice, people are forced to unmount any filesystems on a USB device before suspending. If the root filesystem is on a USB device, the system can't be suspended at all. (All right, it _can_ be suspended – but it will crash as soon as it wakes up, which isn't much better.)

What is the solution?

The kernel includes a feature called USB-persist. It tries to work around these issues by allowing the core USB device data structures to persist across a power-session disruption.

It works like this. If the kernel sees that a USB host controller is not in the expected state during resume (i.e., if the controller was reset or otherwise had lost power) then it applies a persistence check to each of the USB devices below that controller for which the "persist" attribute is set. It doesn't try to resume the device; that can't work once the power session is gone. Instead it issues a USB port reset and then re-enumerates the device. (This is exactly the same thing that happens whenever a USB device is reset.) If the re-enumeration shows that the device now attached to that port has the same descriptors as before, including the Vendor and Product IDs, then the kernel continues to use the same device structure. In effect, the kernel treats the device as though it had merely been reset instead of unplugged.

The same thing happens if the host controller is in the expected state but a USB device was unplugged and then replugged, or if a USB device fails to carry out a normal resume.

If no device is now attached to the port, or if the descriptors are different from what the kernel remembers, then the treatment is what you would expect. The kernel destroys the old device structure and behaves as though the old device had been unplugged and a new device plugged in.

The end result is that the USB device remains available and usable. Filesystem mounts and memory mappings are unaffected, and the world is now a good and happy place.

Note that the "USB-persist" feature will be applied only to those devices for which it is enabled. You can enable the feature by doing (as root):

```
echo 1 >/sys/bus/usb/devices/.../power/persist
```

where the "..." should be filled in the with the device's ID. Disable the feature by writing 0 instead of 1. For hubs the feature is automatically and permanently enabled and the power/persist file doesn't even exist, so you only have to worry about setting it for devices where it really matters.

Is this the best solution?

Perhaps not. Arguably, keeping track of mounted filesystems and memory mappings across device disconnects should be handled by a centralized Logical Volume Manager. Such a solution would allow you to plug in a USB flash device, create a persistent volume associated with it, unplug the flash device, plug it back in later, and still have the same persistent volume associated with the device. As such it would be more far-reaching than USB-persist.

On the other hand, writing a persistent volume manager would be a big job and using it would require significant input from the user. This solution is much quicker and easier – and it exists now, a giant point in its favor!

Furthermore, the USB-persist feature applies to _all_ USB devices, not just mass-storage devices. It might turn out to be equally useful for other device types, such as network interfaces.

WARNING: USB-persist can be dangerous!!

When recovering an interrupted power session the kernel does its best to make sure the USB device hasn't been changed; that is, the same device is still plugged into the port as before. But the checks aren't guaranteed to be 100% accurate.

If you replace one USB device with another of the same type (same manufacturer, same IDs, and so on) there's an excellent chance the kernel won't detect the change. The serial number string and other descriptors are compared with the kernel's stored values, but this might not help since manufacturers frequently omit serial numbers entirely in their devices.

Furthermore it's quite possible to leave a USB device exactly the same while changing its media. If you replace the flash memory card in a USB card reader while the system is asleep, the kernel will have no way to know you did it. The kernel will assume that nothing has happened and will continue to use the partition tables, inodes, and memory mappings for the old card.

If the kernel gets fooled in this way, it's almost certain to cause data corruption and to crash your system. You'll have no one to blame but yourself.

For those devices with avoid_reset_quirk attribute being set, persist maybe fail because they may morph after reset.

YOU HAVE BEEN WARNED! USE AT YOUR OWN RISK!

That having been said, most of the time there shouldn't be any trouble at all. The USB-persist feature can be extremely useful. Make the most of it.

USB Error codes

Revised 2004-Oct-21

This is the documentation of (hopefully) all possible error codes (and their interpretation) that can be returned from usbcore.

Some of them are returned by the Host Controller Drivers (HCDs), which device drivers only see through usbcore. As a rule, all the HCDs should behave the same except for transfer speed dependent behaviors and the way certain faults are reported.

Error codes returned by `usb_submit_urb()`

Non-USB-specific:

0	URB submission went fine
`-ENOMEM`	no memory for allocation of internal structures

USB-specific:

`-EBUSY`	The URB is already active.
`-ENODEV`	specified USB-device or bus doesn't exist
`-ENOENT`	specified interface or endpoint does not exist or is not enabled
`-ENXIO`	host controller driver does not support queuing of this type of urb. (treat as a host controller bug.)
`-EINVAL`	1. Invalid transfer type specified (or not supported) 2. Invalid or unsupported periodic transfer interval 3. ISO: attempted to change transfer interval 4. ISO: `number_of_packets` is < 0 5. various other cases
`-EXDEV`	ISO: `URB_ISO_ASAP` wasn't specified and all the frames the URB would be scheduled in have already expired.
`-EFBIG`	Host controller driver can't schedule that many ISO frames.
`-EPIPE`	The pipe type specified in the URB doesn't match the endpoint's actual type.
`-EMSGSIZE`	1. endpoint maxpacket size is zero; it is not usable in the current interface altsetting. 2. ISO packet is larger than the endpoint maxpacket. 3. requested data transfer length is invalid: negative or too large for the host controller.
`-ENOSPC`	This request would overcommit the usb bandwidth reserved for periodic transfers (interrupt, isochronous).
`-ESHUTDOWN`	The device or host controller has been disabled due to some problem that could not be worked around.
`-EPERM`	Submission failed because `urb->reject` was set.
`-EHOSTUNREACH`	URB was rejected because the device is suspended.
`-ENOEXEC`	A control URB doesn't contain a Setup packet.

Error codes returned by in `urb->status` or in `iso_frame_desc[n].status` (for ISO)

USB device drivers may only test urb status values in completion handlers. This is because otherwise there would be a race between HCDs updating these values on one CPU, and device drivers testing them on another CPU.

A transfer's actual_length may be positive even when an error has been reported. That's because transfers often involve several packets, so that one or more packets could finish before an error stops further endpoint I/O.

For isochronous URBs, the urb status value is non-zero only if the URB is unlinked, the device is removed, the host controller is disabled, or the total transferred length is less than the requested length and the `URB_SHORT_NOT_OK` flag is set. Completion handlers for isochronous URBs should only see `urb->status` set to zero, `-ENOENT`, `-ECONNRESET`, `-ESHUTDOWN`, or `-EREMOTEIO`. Individual frame descriptor status fields may report more status codes.

`0`	Transfer completed successfully
`-ENOENT`	URB was synchronously unlinked by *`usb_unlink_urb()`*
`-EINPROGRESS`	URB still pending, no results yet (That is, if drivers see this it's a bug.)
`-EPROTO` [1], [2]	1. bitstuff error 2. no response packet received within the prescribed bus turn-around time 3. unknown USB error
`-EILSEQ` [1], [2]	1. CRC mismatch 2. no response packet received within the prescribed bus turn-around time 3. unknown USB error Note that often the controller hardware does not distinguish among cases a), b), and c), so a driver cannot tell whether there was a protocol error, a failure to respond (often caused by device disconnect), or some other fault.
`-ETIME` [2]	No response packet received within the prescribed bus turn-around time. This error may instead be reported as `-EPROTO` or `-EILSEQ`.
`-ETIMEDOUT`	Synchronous USB message functions use this code to indicate timeout expired before the transfer completed, and no other error was reported by HC.
`-EPIPE` [2]	Endpoint stalled. For non-control endpoints, reset this status with *`usb_clear_halt()`*.
`-ECOMM`	During an IN transfer, the host controller received data from an endpoint faster than it could be written to system memory
`-ENOSR`	During an OUT transfer, the host controller could not retrieve data from system memory fast enough to keep up with the USB data rate
`-EOVERFLOW` [1]	The amount of data returned by the endpoint was greater than either the max packet size of the endpoint or the remaining buffer size. "Babble".
`-EREMOTEIO`	The data read from the endpoint did not fill the specified buffer, and `URB_SHORT_NOT_OK` was set in `urb->transfer_flags`.
`-ENODEV`	Device was removed. Often preceded by a burst of other errors, since the hub driver doesn't detect device removal events immediately.
`-EXDEV`	ISO transfer only partially completed (only set in `iso_frame_desc[n].status`, not `urb->status`)
`-EINVAL`	ISO madness, if this happens: Log off and go home
`-ECONNRESET`	URB was asynchronously unlinked by *`usb_unlink_urb()`*
`-ESHUTDOWN`	The device or host controller has been disabled due to some problem that could not be worked around, such as a physical disconnect.

[1] Error codes like `-EPROTO`, `-EILSEQ` and `-EOVERFLOW` normally indicate hardware problems such as bad devices (including firmware) or cables.

[2] This is also one of several codes that different kinds of host controller use to indicate a transfer has failed because of device disconnect. In the interval before the hub driver starts disconnect processing, devices may receive such fault reports for every request.

Error codes returned by usbcore-functions

Note:

expect also other submit and transfer status codes

usb_register():

-EINVAL	error during registering new driver

usb_get_*/usb_set_*(), *usb_control_msg()*, *usb_bulk_msg()*:

-ETIMEDOUT	Timeout expired before the transfer completed.

Writing USB Device Drivers

Author Greg Kroah-Hartman

Introduction

The Linux USB subsystem has grown from supporting only two different types of devices in the 2.2.7 kernel (mice and keyboards), to over 20 different types of devices in the 2.4 kernel. Linux currently supports almost all USB class devices (standard types of devices like keyboards, mice, modems, printers and speakers) and an ever-growing number of vendor-specific devices (such as USB to serial converters, digital cameras, Ethernet devices and MP3 players). For a full list of the different USB devices currently supported, see Resources.

The remaining kinds of USB devices that do not have support on Linux are almost all vendor-specific devices. Each vendor decides to implement a custom protocol to talk to their device, so a custom driver usually needs to be created. Some vendors are open with their USB protocols and help with the creation of Linux drivers, while others do not publish them, and developers are forced to reverse-engineer. See Resources for some links to handy reverse-engineering tools.

Because each different protocol causes a new driver to be created, I have written a generic USB driver skeleton, modelled after the pci-skeleton.c file in the kernel source tree upon which many PCI network drivers have been based. This USB skeleton can be found at drivers/usb/usb-skeleton.c in the kernel source tree. In this article I will walk through the basics of the skeleton driver, explaining the different pieces and what needs to be done to customize it to your specific device.

Linux USB Basics

If you are going to write a Linux USB driver, please become familiar with the USB protocol specification. It can be found, along with many other useful documents, at the USB home page (see Resources). An excellent introduction to the Linux USB subsystem can be found at the USB Working Devices List (see Resources). It explains how the Linux USB subsystem is structured and introduces the reader to the concept of USB urbs (USB Request Blocks), which are essential to USB drivers.

The first thing a Linux USB driver needs to do is register itself with the Linux USB subsystem, giving it some information about which devices the driver supports and which functions to call when a device supported by the driver is inserted or removed from the system. All of this information is passed to the USB subsystem in the *usb_driver* structure. The skeleton driver declares a *usb_driver* as:

```
static struct usb_driver skel_driver = {
        .name        = "skeleton",
        .probe       = skel_probe,
        .disconnect  = skel_disconnect,
        .fops           = &skel_fops,
        .minor          = USB_SKEL_MINOR_BASE,
        .id_table       = skel_table,
};
```

The variable name is a string that describes the driver. It is used in informational messages printed to the system log. The probe and disconnect function pointers are called when a device that matches the information provided in the `id_table` variable is either seen or removed.

The fops and minor variables are optional. Most USB drivers hook into another kernel subsystem, such as the SCSI, network or TTY subsystem. These types of drivers register themselves with the other kernel subsystem, and any user-space interactions are provided through that interface. But for drivers that do not have a matching kernel subsystem, such as MP3 players or scanners, a method of interacting with user space is needed. The USB subsystem provides a way to register a minor device number and a set of `file_operations` function pointers that enable this user-space interaction. The skeleton driver needs this kind of interface, so it provides a minor starting number and a pointer to its `file_operations` functions.

The USB driver is then registered with a call to `usb_register()`, usually in the driver's init function, as shown here:

```
static int __init usb_skel_init(void)
{
        int result;

        /* register this driver with the USB subsystem */
        result = usb_register(&skel_driver);
        if (result < 0) {
                err("usb_register failed for the "__FILE__ "driver."
                    "Error number %d", result);
                return -1;
        }

        return 0;
}
module_init(usb_skel_init);
```

When the driver is unloaded from the system, it needs to deregister itself with the USB subsystem. This is done with the *`usb_deregister()`* function:

```
static void __exit usb_skel_exit(void)
{
        /* deregister this driver with the USB subsystem */
        usb_deregister(&skel_driver);
}
module_exit(usb_skel_exit);
```

To enable the linux-hotplug system to load the driver automatically when the device is plugged in, you need to create a `MODULE_DEVICE_TABLE`. The following code tells the hotplug scripts that this module supports a single device with a specific vendor and product ID:

```
/* table of devices that work with this driver */
static struct usb_device_id skel_table [] = {
        { USB_DEVICE(USB_SKEL_VENDOR_ID, USB_SKEL_PRODUCT_ID) },
        { }                      /* Terminating entry */
};
MODULE_DEVICE_TABLE (usb, skel_table);
```

There are other macros that can be used in describing a struct *`usb_device_id`* for drivers that support a whole class of USB drivers. See *usb.h* for more information on this.

Device operation

When a device is plugged into the USB bus that matches the device ID pattern that your driver registered with the USB core, the probe function is called. The *usb_device* structure, interface number and the interface ID are passed to the function:

```
static int skel_probe(struct usb_interface *interface,
    const struct usb_device_id *id)
```

The driver now needs to verify that this device is actually one that it can accept. If so, it returns 0. If not, or if any error occurs during initialization, an errorcode (such as -ENOMEM or -ENODEV) is returned from the probe function.

In the skeleton driver, we determine what end points are marked as bulk-in and bulk-out. We create buffers to hold the data that will be sent and received from the device, and a USB urb to write data to the device is initialized.

Conversely, when the device is removed from the USB bus, the disconnect function is called with the device pointer. The driver needs to clean any private data that has been allocated at this time and to shut down any pending urbs that are in the USB system.

Now that the device is plugged into the system and the driver is bound to the device, any of the functions in the file_operations structure that were passed to the USB subsystem will be called from a user program trying to talk to the device. The first function called will be open, as the program tries to open the device for I/O. We increment our private usage count and save a pointer to our internal structure in the file structure. This is done so that future calls to file operations will enable the driver to determine which device the user is addressing. All of this is done with the following code:

```
/* increment our usage count for the module */
++skel->open_count;

/* save our object in the file's private structure */
file->private_data = dev;
```

After the open function is called, the read and write functions are called to receive and send data to the device. In the skel_write function, we receive a pointer to some data that the user wants to send to the device and the size of the data. The function determines how much data it can send to the device based on the size of the write urb it has created (this size depends on the size of the bulk out end point that the device has). Then it copies the data from user space to kernel space, points the urb to the data and submits the urb to the USB subsystem. This can be seen in the following code:

```
/* we can only write as much as 1 urb will hold */
bytes_written = (count > skel->bulk_out_size) ? skel->bulk_out_size : count;

/* copy the data from user space into our urb */
copy_from_user(skel->write_urb->transfer_buffer, buffer, bytes_written);

/* set up our urb */
usb_fill_bulk_urb(skel->write_urb,
                  skel->dev,
                  usb_sndbulkpipe(skel->dev, skel->bulk_out_endpointAddr),
                  skel->write_urb->transfer_buffer,
                  bytes_written,
                  skel_write_bulk_callback,
                  skel);

/* send the data out the bulk port */
result = usb_submit_urb(skel->write_urb);
if (result) {
        err("Failed submitting write urb, error %d", result);
}
```

When the write urb is filled up with the proper information using the *usb_fill_bulk_urb()* function, we point the urb's completion callback to call our own skel_write_bulk_callback function. This function is called when the urb is finished by the USB subsystem. The callback function is called in interrupt context, so caution must be taken not to do very much processing at that time. Our implementation of skel_write_bulk_callback merely reports if the urb was completed successfully or not and then returns.

The read function works a bit differently from the write function in that we do not use an urb to transfer data from the device to the driver. Instead we call the *usb_bulk_msg()* function, which can be used to send or receive data from a device without having to create urbs and handle urb completion callback functions. We call the *usb_bulk_msg()* function, giving it a buffer into which to place any data received from the device and a timeout value. If the timeout period expires without receiving any data from the device, the function will fail and return an error message. This can be shown with the following code:

```
/* do an immediate bulk read to get data from the device */
retval = usb_bulk_msg (skel->dev,
                       usb_rcvbulkpipe (skel->dev,
                       skel->bulk_in_endpointAddr),
                       skel->bulk_in_buffer,
                       skel->bulk_in_size,
                       &count, HZ*10);
/* if the read was successful, copy the data to user space */
if (!retval) {
        if (copy_to_user (buffer, skel->bulk_in_buffer, count))
                retval = -EFAULT;
        else
                retval = count;
}
```

The *usb_bulk_msg()* function can be very useful for doing single reads or writes to a device; however, if you need to read or write constantly to a device, it is recommended to set up your own urbs and submit them to the USB subsystem.

When the user program releases the file handle that it has been using to talk to the device, the release function in the driver is called. In this function we decrement our private usage count and wait for possible pending writes:

```
/* decrement our usage count for the device */
--skel->open_count;
```

One of the more difficult problems that USB drivers must be able to handle smoothly is the fact that the USB device may be removed from the system at any point in time, even if a program is currently talking to it. It needs to be able to shut down any current reads and writes and notify the user-space programs that the device is no longer there. The following code (function skel_delete) is an example of how to do this:

```
static inline void skel_delete (struct usb_skel *dev)
{
    kfree (dev->bulk_in_buffer);
    if (dev->bulk_out_buffer != NULL)
        usb_free_coherent (dev->udev, dev->bulk_out_size,
            dev->bulk_out_buffer,
            dev->write_urb->transfer_dma);
    usb_free_urb (dev->write_urb);
    kfree (dev);
}
```

If a program currently has an open handle to the device, we reset the flag device_present. For every read, write, release and other functions that expect a device to be present, the driver first checks this flag to see if the device is still present. If not, it releases that the device has disappeared, and a -ENODEV error is returned to the user-space program. When the release function is eventually called, it determines if there is no device and if not, it does the cleanup that the skel_disconnect function normally does if there are no open files on the device (see Listing 5).

Isochronous Data

This usb-skeleton driver does not have any examples of interrupt or isochronous data being sent to or from the device. Interrupt data is sent almost exactly as bulk data is, with a few minor exceptions. Isochronous data works differently with continuous streams of data being sent to or from the device. The audio and video camera drivers are very good examples of drivers that handle isochronous data and will be useful if you also need to do this.

Conclusion

Writing Linux USB device drivers is not a difficult task as the usb-skeleton driver shows. This driver, combined with the other current USB drivers, should provide enough examples to help a beginning author create a working driver in a minimal amount of time. The linux-usb-devel mailing list archives also contain a lot of helpful information.

Resources

The Linux USB Project: http://www.linux-usb.org/

Linux Hotplug Project: http://linux-hotplug.sourceforge.net/

Linux USB Working Devices List: http://www.qbik.ch/usb/devices/

linux-usb-devel Mailing List Archives: http://marc.theaimsgroup.com/?l=linux-usb-devel

Programming Guide for Linux USB Device Drivers: http://lmu.web.psi.ch/docu/manuals/software_manuals/linux_sl

USB Home Page: http://www.usb.org

Synopsys DesignWare Core SuperSpeed USB 3.0 Controller

Author Felipe Balbi <felipe.balbi@linux.intel.com>

Date April 2017

Introduction

The *Synopsys DesignWare Core SuperSpeed USB 3.0 Controller* (hereinafter referred to as *DWC3*) is a USB SuperSpeed compliant controller which can be configured in one of 4 ways:

1. Peripheral-only configuration
2. Host-only configuration
3. Dual-Role configuration
4. Hub configuration

Linux currently supports several versions of this controller. In all likelyhood, the version in your SoC is already supported. At the time of this writing, known tested versions range from 2.02a to 3.10a. As a rule of thumb, anything above 2.02a should work reliably well.

Currently, we have many known users for this driver. In alphabetical order:

1. Cavium
2. Intel Corporation
3. Qualcomm
4. Rockchip

5. ST
6. Samsung
7. Texas Instruments
8. Xilinx

Summary of Features

For details about features supported by your version of DWC3, consult your IP team and/or *Synopsys DesignWare Core SuperSpeed USB 3.0 Controller Databook*. Following is a list of features supported by the driver at the time of this writing:

1. Up to 16 bidirectional endpoints (including the control pipe - ep0)
2. Flexible endpoint configuration
3. Simultaneous IN and OUT transfer support
4. Scatter-list support
5. Up to 256 TRBs [1] per endpoint
6. Support for all transfer types (*Control*, *Bulk*, *Interrupt*, and *Isochronous*)
7. SuperSpeed Bulk Streams
8. Link Power Management
9. Trace Events for debugging
10. DebugFS [3] interface

These features have all been exercised with many of the **in-tree** gadget drivers. We have verified both *ConfigFS* [4] and legacy gadget drivers.

Driver Design

The DWC3 driver sits on the *drivers/usb/dwc3/* directory. All files related to this driver are in this one directory. This makes it easy for new-comers to read the code and understand how it behaves.

Because of DWC3's configuration flexibility, the driver is a little complex in some places but it should be rather straightforward to understand.

The biggest part of the driver refers to the Gadget API.

Known Limitations

Like any other HW, DWC3 has its own set of limitations. To avoid constant questions about such problems, we decided to document them here and have a single location to where we could point users.

OUT Transfer Size Requirements

According to Synopsys Databook, all OUT transfer TRBs [1] must have their *size* field set to a value which is integer divisible by the endpoint's *wMaxPacketSize*. This means that *e.g.* in order to receive a Mass Storage *CBW* [5], req->length must either be set to a value that's divisible by *wMaxPacketSize* (1024 on SuperSpeed, 512 on HighSpeed, etc), or DWC3 driver must add a Chained TRB pointing to a throw-away buffer for the remaining length. Without this, OUT transfers will **NOT** start.

[1] Transfer Request Block
[3] The Debug File System
[4] The Config File System
[5] Command Block Wrapper

Note that as of this writing, this won't be a problem because DWC3 is fully capable of appending a chained TRB for the remaining length and completely hide this detail from the gadget driver. It's still worth mentioning because this seems to be the largest source of queries about DWC3 and *non-working transfers*.

TRB Ring Size Limitation

We, currently, have a hard limit of 256 TRBs [1] per endpoint, with the last TRB being a Link TRB [2] pointing back to the first. This limit is arbitrary but it has the benefit of adding up to exactly 4096 bytes, or 1 Page.

DWC3 driver will try its best to cope with more than 255 requests and, for the most part, it should work normally. However this is not something that has been exercised very frequently. If you experience any problems, see section **Reporting Bugs** below.

Reporting Bugs

Whenever you encounter a problem with DWC3, first and foremost you should make sure that:

1. You're running latest tag from Linus' tree
2. You can reproduce the error without any out-of-tree changes to DWC3
3. You have checked that it's not a fault on the host machine

After all these are verified, then here's how to capture enough information so we can be of any help to you.

Required Information

DWC3 relies exclusively on Trace Events for debugging. Everything is exposed there, with some extra bits being exposed to DebugFS [3].

In order to capture DWC3's Trace Events you should run the following commands **before** plugging the USB cable to a host machine:

```
# mkdir -p /d
# mkdir -p /t
# mount -t debugfs none /d
# mount -t tracefs none /t
# echo 81920 > /t/buffer_size_kb
# echo 1 > /t/events/dwc3/enable
```

After this is done, you can connect your USB cable and reproduce the problem. As soon as the fault is reproduced, make a copy of files `trace` and `regdump`, like so:

```
# cp /t/trace /root/trace.txt
# cat /d/*dwc3*/regdump > /root/regdump.txt
```

Make sure to compress `trace.txt` and `regdump.txt` in a tarball and email it to me with linux-usb in Cc. If you want to be extra sure that I'll help you, write your subject line in the following format:

> **[BUG REPORT] usb: dwc3: Bug while doing XYZ**

On the email body, make sure to detail what you doing, which gadget driver you were using, how to reproduce the problem, what SoC you're using, which OS (and its version) was running on the Host machine.

With all this information, we should be able to understand what's going on and be helpful to you.

[2] Transfer Request Block pointing to another Transfer Request Block.

Debugging

First and foremost a disclaimer:

```
DISCLAIMER: The information available on DebugFS and/or TraceFS can
change at any time at any Major Linux Kernel Release. If writing
scripts, do **NOT** assume information to be available in the
current format.
```

With that out of the way, let's carry on.

If you're willing to debug your own problem, you deserve a round of applause :-)

Anyway, there isn't much to say here other than Trace Events will be really helpful in figuring out issues with DWC3. Also, access to Synopsys Databook will be **really** valuable in this case.

A USB Sniffer can be helpful at times but it's not entirely required, there's a lot that can be understood without looking at the wire.

Feel free to email me and Cc linux-usb if you need any help.

DebugFS

DebugFS is very good for gathering snapshots of what's going on with DWC3 and/or any endpoint.

On DWC3's DebugFS directory, you will find the following files and directories:

`ep[0..15]{in,out}/` `link_state` `regdump` `testmode`

`link_state`

When read, `link_state` will print out one of `U0`, `U1`, `U2`, `U3`, `SS.Disabled`, `RX.Detect`, `SS.Inactive`, `Polling`, `Recovery`, `Hot Reset`, `Compliance`, `Loopback`, `Reset`, `Resume` or `UNKNOWN link state`.

This file can also be written to in order to force link to one of the states above.

`regdump`

File name is self-explanatory. When read, `regdump` will print out a register dump of DWC3. Note that this file can be grepped to find the information you want.

`testmode`

When read, `testmode` will print out a name of one of the specified USB 2.0 Testmodes (`test_j`, `test_k`, `test_se0_nak`, `test_packet`, `test_force_enable`) or the string `no test` in case no tests are currently being executed.

In order to start any of these test modes, the same strings can be written to the file and DWC3 will enter the requested test mode.

`ep[0..15]{in,out}`

For each endpoint we expose one directory following the naming convention `ep$num$dir` *(ep0in, ep0out, ep1in, ...)*. Inside each of these directories you will find the following files:

`descriptor_fetch_queue` `event_queue` `rx_fifo_queue` `rx_info_queue` `rx_request_queue` `transfer_type` `trb_ring` `tx_fifo_queue` `tx_request_queue`

With access to Synopsys Databook, you can decode the information on them.

`transfer_type` When read, `transfer_type` will print out one of `control`, `bulk`, `interrupt` or `isochronous` depending on what the endpoint descriptor says. If the endpoint hasn't been enabled yet, it will print `--`.

`trb_ring` When read, `trb_ring` will print out details about all TRBs on the ring. It will also tell you where our enqueue and dequeue pointers are located in the ring:

```
buffer_addr,size,type,ioc,isp_imi,csp,chn,lst,hwo
000000002c754000,481,normal,1,0,1,0,0,0
000000002c75c000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
000000002c78c000,481,normal,1,0,1,0,0,0
000000002c754000,481,normal,1,0,1,0,0,0
000000002c75c000,481,normal,1,0,1,0,0,0
000000002c784000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
000000002c78c000,481,normal,1,0,1,0,0,0
000000002c790000,481,normal,1,0,1,0,0,0
000000002c758000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
000000002c790000,481,normal,1,0,1,0,0,0
000000002c758000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c784000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
000000002c78c000,481,normal,1,0,1,0,0,0
000000002c754000,481,normal,1,0,1,0,0,0
000000002c758000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c784000,481,normal,1,0,1,0,0,0
000000002c78c000,481,normal,1,0,1,0,0,0
000000002c790000,481,normal,1,0,1,0,0,0
000000002c758000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
000000002c790000,481,normal,1,0,1,0,0,0
000000002c758000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
000000002c790000,481,normal,1,0,1,0,0,0
000000002c758000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
000000002c790000,481,normal,1,0,1,0,0,0
000000002c758000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
000000002c790000,481,normal,1,0,1,0,0,0
000000002c758000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
000000002c790000,481,normal,1,0,1,0,0,0
000000002c758000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
000000002c790000,481,normal,1,0,1,0,0,0
000000002c758000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
000000002c790000,481,normal,1,0,1,0,0,0
```

```
000000002c758000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c78c000,481,normal,1,0,1,0,0,0
000000002c784000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
000000002c78c000,481,normal,1,0,1,0,0,0
000000002c754000,481,normal,1,0,1,0,0,0
000000002c758000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
000000002c790000,481,normal,1,0,1,0,0,0
000000002c758000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c758000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c78c000,481,normal,1,0,1,0,0,0
000000002c75c000,481,normal,1,0,1,0,0,0
000000002c78c000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c754000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
000000002c754000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
000000002c78c000,481,normal,1,0,1,0,0,0
000000002c790000,481,normal,1,0,1,0,0,0
000000002c754000,481,normal,1,0,1,0,0,0
000000002c758000,481,normal,1,0,1,0,0,0
000000002c75c000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c784000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
000000002c78c000,481,normal,1,0,1,0,0,0
000000002c790000,481,normal,1,0,1,0,0,0
000000002c754000,481,normal,1,0,1,0,0,0
000000002c758000,481,normal,1,0,1,0,0,0
000000002c75c000,512,normal,1,0,1,0,0,1        D
0000000000000000,0,UNKNOWN,0,0,0,0,0,0       E
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
```

```
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
```

```
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
```

```
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
00000000381ab000,0,link,0,0,0,0,0,1
```

Trace Events

DWC3 also provides several trace events which help us gathering information about the behavior of the driver during runtime.

In order to use these events, you must enable `CONFIG_FTRACE` in your kernel config.

For details about how enable DWC3 events, see section **Reporting Bugs**.

The following subsections will give details about each Event Class and each Event defined by DWC3.

MMIO

It is sometimes useful to look at every MMIO access when looking for bugs. Because of that, DWC3 offers two Trace Events (one for dwc3_readl() and one for dwc3_writel()). `TP_printk` follows:

```
TP_printk("addr %p value %08x", __entry->base + __entry->offset,
              __entry->value)
```

Interrupt Events

Every IRQ event can be logged and decoded into a human readable string. Because every event will be different, we don't give an example other than the `TP_printk` format used:

```
TP_printk("event (%08x): %s", __entry->event,
              dwc3_decode_event(__entry->event, __entry->ep0state))
```

Control Request

Every USB Control Request can be logged to the trace buffer. The output format is:

```
TP_printk("%s", dwc3_decode_ctrl(__entry->bRequestType,
                              __entry->bRequest, __entry->wValue,
                              __entry->wIndex, __entry->wLength)
)
```

Note that Standard Control Requests will be decoded into human-readable strings with their respective arguments. Class and Vendor requests will be printed out a sequence of 8 bytes in hex format.

Lifetime of a `struct usb_request`

The entire lifetime of a `struct usb_request` can be tracked on the trace buffer. We have one event for each of allocation, free, queueing, dequeueing, and giveback. Output format is:

```
TP_printk("%s: req %p length %u/%u %s%s%s ==> %d",
      __get_str(name), __entry->req, __entry->actual, __entry->length,
      __entry->zero ? "Z" : "z",
      __entry->short_not_ok ? "S" : "s",
      __entry->no_interrupt ? "i" : "I",
      __entry->status
)
```

Generic Commands

We can log and decode every Generic Command with its completion code. Format is:

```
TP_printk("cmd '%s' [%x] param %08x --> status: %s",
      dwc3_gadget_generic_cmd_string(__entry->cmd),
      __entry->cmd, __entry->param,
      dwc3_gadget_generic_cmd_status_string(__entry->status)
)
```

Endpoint Commands

Endpoints commands can also be logged together with completion code. Format is:

```
TP_printk("%s: cmd '%s' [%d] params %08x %08x %08x --> status: %s",
      __get_str(name), dwc3_gadget_ep_cmd_string(__entry->cmd),
      __entry->cmd, __entry->param0,
      __entry->param1, __entry->param2,
      dwc3_ep_cmd_status_string(__entry->cmd_status)
)
```

Lifetime of a TRB

A TRB Lifetime is simple. We are either preparing a TRB or completing it. With these two events, we can see how a TRB changes over time. Format is:

```
TP_printk("%s: %d/%d trb %p buf %08x%08x size %s%d ctrl %08x (%c%c%c%c:%c%c:%s)",
      __get_str(name), __entry->queued, __entry->allocated,
      __entry->trb, __entry->bph, __entry->bpl,
      ({char *s;
      int pcm = ((__entry->size >> 24) & 3) + 1;
      switch (__entry->type) {
      case USB_ENDPOINT_XFER_INT:
      case USB_ENDPOINT_XFER_ISOC:
              switch (pcm) {
              case 1:
                      s = "1x ";
                      break;
              case 2:
                      s = "2x ";
                      break;
              case 3:
                      s = "3x ";
                      break;
              }
      default:
              s = "";
      } s; }),
      DWC3_TRB_SIZE_LENGTH(__entry->size), __entry->ctrl,
      __entry->ctrl & DWC3_TRB_CTRL_HWO ? 'H' : 'h',
      __entry->ctrl & DWC3_TRB_CTRL_LST ? 'L' : 'l',
      __entry->ctrl & DWC3_TRB_CTRL_CHN ? 'C' : 'c',
      __entry->ctrl & DWC3_TRB_CTRL_CSP ? 'S' : 's',
      __entry->ctrl & DWC3_TRB_CTRL_ISP_IMI ? 'S' : 's',
      __entry->ctrl & DWC3_TRB_CTRL_IOC ? 'C' : 'c',
    dwc3_trb_type_string(DWC3_TRBCTL_TYPE(__entry->ctrl))
)
```

Lifetime of an Endpoint

And endpoint's lifetime is summarized with enable and disable operations, both of which can be traced. Format is:

```
TP_printk("%s: mps %d/%d streams %d burst %d ring %d/%d flags %c:%c%c%c%c%c:%c:%c",
      __get_str(name), __entry->maxpacket,
      __entry->maxpacket_limit, __entry->max_streams,
      __entry->maxburst, __entry->trb_enqueue,
      __entry->trb_dequeue,
      __entry->flags & DWC3_EP_ENABLED ? 'E' : 'e',
      __entry->flags & DWC3_EP_STALL ? 'S' : 's',
      __entry->flags & DWC3_EP_WEDGE ? 'W' : 'w',
      __entry->flags & DWC3_EP_BUSY ? 'B' : 'b',
      __entry->flags & DWC3_EP_PENDING_REQUEST ? 'P' : 'p',
      __entry->flags & DWC3_EP_MISSED_ISOC ? 'M' : 'm',
      __entry->flags & DWC3_EP_END_TRANSFER_PENDING ? 'E' : 'e',
      __entry->direction ? '<' : '>'
)
```

Structures, Methods and Definitions

struct **dwc3_event_buffer**
Software event buffer representation

Definition

```
struct dwc3_event_buffer {
  void *buf;
  void *cache;
  unsigned length;
  unsigned int            lpos;
  unsigned int            count;
  unsigned int            flags;
#define DWC3_EVENT_PENDING      BIT(0);
  dma_addr_t dma;
  struct dwc3             *dwc;
};
```

Members

buf _THE_ buffer

cache The buffer cache used in the threaded interrupt

length size of this buffer

lpos event offset

count cache of last read event count register

flags flags related to this event buffer

dma dma_addr_t

dwc pointer to DWC controller

struct **dwc3_ep**
device side endpoint representation

Definition

```
struct dwc3_ep {
  struct usb_ep           endpoint;
  struct list_head        pending_list;
  struct list_head        started_list;
  wait_queue_head_t wait_end_transfer;
  spinlock_t lock;
  void __iomem            *regs;
  struct dwc3_trb         *trb_pool;
  dma_addr_t trb_pool_dma;
  struct dwc3             *dwc;
  u32 saved_state;
  unsigned flags;
#define DWC3_EP_ENABLED         BIT(0);
#define DWC3_EP_STALL           BIT(1);
#define DWC3_EP_WEDGE           BIT(2);
#define DWC3_EP_BUSY            BIT(4);
#define DWC3_EP_PENDING_REQUEST BIT(5);
#define DWC3_EP_MISSED_ISOC     BIT(6);
#define DWC3_EP_END_TRANSFER_PENDING    BIT(7);
#define DWC3_EP_TRANSFER_STARTED BIT(8);
#define DWC3_EP0_DIR_IN         BIT(31);
  u8 trb_enqueue;
  u8 trb_dequeue;
  u8 number;
  u8 type;
  u8 resource_index;
  u32 allocated_requests;
  u32 queued_requests;
  u32 frame_number;
  u32 interval;
  char name[20];
  unsigned direction:1;
  unsigned stream_capable:1;
};
```

Members

endpoint usb endpoint

pending_list list of pending requests for this endpoint

started_list list of started requests on this endpoint

wait_end_transfer wait_queue_head_t for waiting on End Transfer complete

lock spinlock for endpoint request queue traversal

regs pointer to first endpoint register

trb_pool array of transaction buffers

trb_pool_dma dma address of **trb_pool**

dwc pointer to DWC controller

saved_state ep state saved during hibernation

flags endpoint flags (wedged, stalled, ...)

trb_enqueue enqueue 'pointer' into TRB array

trb_dequeue dequeue 'pointer' into TRB array

number endpoint number (1 - 15)

type set to bmAttributes & USB_ENDPOINT_XFERTYPE_MASK

resource_index Resource transfer index

allocated_requests number of requests allocated

queued_requests number of requests queued for transfer

frame_number set to the frame number we want this transfer to start (ISOC)

interval the interval on which the ISOC transfer is started

name a human readable name e.g. ep1out-bulk

direction true for TX, false for RX

stream_capable true when streams are enabled

struct **dwc3_trb**
transfer request block (hw format)

Definition

```
struct dwc3_trb {
  u32 bpl;
  u32 bph;
  u32 size;
  u32 ctrl;
};
```

Members

bpl DW0-3

bph DW4-7

size DW8-B

ctrl DWC-F

struct **dwc3_hwparams**
copy of HWPARAMS registers

Definition

```
struct dwc3_hwparams {
  u32 hwparams0;
  u32 hwparams1;
  u32 hwparams2;
  u32 hwparams3;
  u32 hwparams4;
  u32 hwparams5;
  u32 hwparams6;
  u32 hwparams7;
  u32 hwparams8;
};
```

Members

hwparams0 GHWPARAMS0

hwparams1 GHWPARAMS1

hwparams2 GHWPARAMS2

hwparams3 GHWPARAMS3

hwparams4 GHWPARAMS4

hwparams5 GHWPARAMS5

hwparams6 GHWPARAMS6

hwparams7 GHWPARAMS7

hwparams8 GHWPARAMS8

struct **dwc3_request**
representation of a transfer request

Definition

```
struct dwc3_request {
  struct usb_request      request;
  struct list_head        list;
  struct dwc3_ep          *dep;
  struct scatterlist      *sg;
  unsigned num_pending_sgs;
  unsigned remaining;
  u8 epnum;
  struct dwc3_trb         *trb;
  dma_addr_t trb_dma;
  unsigned unaligned:1;
  unsigned direction:1;
  unsigned mapped:1;
  unsigned started:1;
  unsigned zero:1;
};
```

Members

request struct usb_request to be transferred

list a list_head used for request queueing

dep struct dwc3_ep owning this request

sg pointer to first incomplete sg

num_pending_sgs counter to pending sgs

remaining amount of data remaining

epnum endpoint number to which this request refers

trb pointer to struct dwc3_trb

trb_dma DMA address of **trb**

unaligned true for OUT endpoints with length not divisible by maxp

direction IN or OUT direction flag

mapped true when request has been dma-mapped

started request is started

zero wants a ZLP

struct **dwc3**
representation of our controller

Definition

```
struct dwc3 {
  struct work_struct      drd_work;
  struct dwc3_trb         *ep0_trb;
  void *bounce;
  void *scratchbuf;
  u8 *setup_buf;
  dma_addr_t ep0_trb_addr;
  dma_addr_t bounce_addr;
  dma_addr_t scratch_addr;
  struct dwc3_request     ep0_usb_req;
  struct completion       ep0_in_setup;
  spinlock_t lock;
  struct device           *dev;
  struct device           *sysdev;
  struct platform_device  *xhci;
  struct resource         xhci_resources[DWC3_XHCI_RESOURCES_NUM];
  struct dwc3_event_buffer *ev_buf;
  struct dwc3_ep          *eps[DWC3_ENDPOINTS_NUM];
  struct usb_gadget       gadget;
  struct usb_gadget_driver *gadget_driver;
  struct usb_phy          *usb2_phy;
  struct usb_phy          *usb3_phy;
  struct phy              *usb2_generic_phy;
  struct phy              *usb3_generic_phy;
  bool phys_ready;
  struct ulpi             *ulpi;
  bool ulpi_ready;
  void __iomem            *regs;
  size_t regs_size;
  enum usb_dr_mode        dr_mode;
  u32 current_dr_role;
  u32 desired_dr_role;
  struct extcon_dev       *edev;
  struct notifier_block   edev_nb;
  enum usb_phy_interface  hsphy_mode;
  u32 fladj;
  u32 irq_gadget;
  u32 nr_scratch;
  u32 u1u2;
  u32 maximum_speed;
  u32 revision;
#define DWC3_REVISION_173A      0x5533173a;
#define DWC3_REVISION_175A      0x5533175a;
#define DWC3_REVISION_180A      0x5533180a;
#define DWC3_REVISION_183A      0x5533183a;
#define DWC3_REVISION_185A      0x5533185a;
#define DWC3_REVISION_187A      0x5533187a;
#define DWC3_REVISION_188A      0x5533188a;
#define DWC3_REVISION_190A      0x5533190a;
#define DWC3_REVISION_194A      0x5533194a;
#define DWC3_REVISION_200A      0x5533200a;
#define DWC3_REVISION_202A      0x5533202a;
#define DWC3_REVISION_210A      0x5533210a;
#define DWC3_REVISION_220A      0x5533220a;
#define DWC3_REVISION_230A      0x5533230a;
#define DWC3_REVISION_240A      0x5533240a;
#define DWC3_REVISION_250A      0x5533250a;
#define DWC3_REVISION_260A      0x5533260a;
#define DWC3_REVISION_270A      0x5533270a;
#define DWC3_REVISION_280A      0x5533280a;
#define DWC3_REVISION_290A      0x5533290a;
#define DWC3_REVISION_300A      0x5533300a;
#define DWC3_REVISION_310A      0x5533310a;
#define DWC3_REVISION_IS_DWC31          0x80000000;
#define DWC3_USB31_REVISION_110A        (0x3131302a | DWC3_REVISION_IS_DWC31);
#define DWC3_USB31_REVISION_120A        (0x3132302a | DWC3_REVISION_IS_DWC31);
  enum dwc3_ep0_next      ep0_next_event;
  enum dwc3_ep0_state     ep0state;
  enum dwc3_link_state    link_state;
  u16 u2sel;
  u16 u2pel;
  u8 u1sel;
  u8 u1pel;
  u8 speed;
  u8 num_eps;
  struct dwc3_hwparams    hwparams;
  struct dentry           *root;
  struct debugfs_regset32 *regset;
  u8 test_mode;
  u8 test_mode_nr;
  u8 lpm_nyet_threshold;
  u8 hird_threshold;
  const char              *hsphy_interface;
  unsigned connected:1;
  unsigned delayed_status:1;
  unsigned ep0_bounced:1;
  unsigned ep0_expect_in:1;
  unsigned has_hibernation:1;
  unsigned sysdev_is_parent:1;
  unsigned has_lpm_erratum:1;
  unsigned is_utmi_l1_suspend:1;
  unsigned is_fpga:1;
  unsigned pending_events:1;
  unsigned pullups_connected:1;
  unsigned setup_packet_pending:1;
  unsigned three_stage_setup:1;
  unsigned usb3_lpm_capable:1;
  unsigned disable_scramble_quirk:1;
  unsigned u2exit_lfps_quirk:1;
  unsigned u2ss_inp3_quirk:1;
  unsigned req_p1p2p3_quirk:1;
  unsigned del_p1p2p3_quirk:1;
  unsigned del_phy_power_chg_quirk:1;
  unsigned lfps_filter_quirk:1;
  unsigned rx_detect_poll_quirk:1;
  unsigned dis_u3_susphy_quirk:1;
  unsigned dis_u2_susphy_quirk:1;
  unsigned dis_enblslpm_quirk:1;
  unsigned dis_rxdet_inp3_quirk:1;
  unsigned dis_u2_freeclk_exists_quirk:1;
  unsigned dis_del_phy_power_chg_quirk:1;
  unsigned dis_tx_ipgap_linecheck_quirk:1;
  unsigned tx_de_emphasis_quirk:1;
  unsigned tx_de_emphasis:2;
  unsigned dis_metastability_quirk:1;
  u16 imod_interval;
};
```

Members

`drd_work` workqueue used for role swapping

`ep0_trb` trb which is used for the ctrl_req

`bounce` address of bounce buffer

`scratchbuf` address of scratch buffer

`setup_buf` used while precessing STD USB requests

`ep0_trb_addr` dma address of **ep0_trb**

`bounce_addr` dma address of **bounce**

`scratch_addr` dma address of scratchbuf

`ep0_usb_req` dummy req used while handling STD USB requests

ep0_in_setup one control transfer is completed and enter setup phase

lock for synchronizing

dev pointer to our struct device

sysdev pointer to the DMA-capable device

xhci pointer to our xHCI child

xhci_resources struct resources for our **xhci** child

ev_buf struct dwc3_event_buffer pointer

eps endpoint array

gadget device side representation of the peripheral controller

gadget_driver pointer to the gadget driver

usb2_phy pointer to USB2 PHY

usb3_phy pointer to USB3 PHY

usb2_generic_phy pointer to USB2 PHY

usb3_generic_phy pointer to USB3 PHY

phys_ready flag to indicate that PHYs are ready

ulpi pointer to ulpi interface

ulpi_ready flag to indicate that ULPI is initialized

regs base address for our registers

regs_size address space size

dr_mode requested mode of operation

current_dr_role current role of operation when in dual-role mode

desired_dr_role desired role of operation when in dual-role mode

edev extcon handle

edev_nb extcon notifier

hsphy_mode UTMI phy mode, one of following: - USBPHY_INTERFACE_MODE_UTMI - USB-PHY_INTERFACE_MODE_UTMIW

fladj frame length adjustment

irq_gadget peripheral controller's IRQ number

nr_scratch number of scratch buffers

u1u2 only used on revisions <1.83a for workaround

maximum_speed maximum speed requested (mainly for testing purposes)

revision revision register contents

ep0_next_event hold the next expected event

ep0state state of endpoint zero

link_state link state

u2sel parameter from Set SEL request.

u2pel parameter from Set SEL request.

u1sel parameter from Set SEL request.

u1pel parameter from Set SEL request.

speed device speed (super, high, full, low)

num_eps number of endpoints

hwparams copy of hwparams registers

root debugfs root folder pointer

regset debugfs pointer to regdump file

test_mode true when we're entering a USB test mode

test_mode_nr test feature selector

lpm_nyet_threshold LPM NYET response threshold

hird_threshold HIRD threshold

hsphy_interface "utmi" or "ulpi"

connected true when we're connected to a host, false otherwise

delayed_status true when gadget driver asks for delayed status

ep0_bounced true when we used bounce buffer

ep0_expect_in true when we expect a DATA IN transfer

has_hibernation true when dwc3 was configured with Hibernation

sysdev_is_parent true when dwc3 device has a parent driver

has_lpm_erratum true when core was configured with LPM Erratum. Note that there's now way for software to detect this in runtime.

is_utmi_l1_suspend the core asserts output signal 0 - utmi_sleep_n 1 - utmi_l1_suspend_n

is_fpga true when we are using the FPGA board

pending_events true when we have pending IRQs to be handled

pullups_connected true when Run/Stop bit is set

setup_packet_pending true when there's a Setup Packet in FIFO. Workaround

three_stage_setup set if we perform a three phase setup

usb3_lpm_capable set if hadrware supports Link Power Management

disable_scramble_quirk set if we enable the disable scramble quirk

u2exit_lfps_quirk set if we enable u2exit lfps quirk

u2ss_inp3_quirk set if we enable P3 OK for U2/SS Inactive quirk

req_p1p2p3_quirk set if we enable request p1p2p3 quirk

del_p1p2p3_quirk set if we enable delay p1p2p3 quirk

del_phy_power_chg_quirk set if we enable delay phy power change quirk

lfps_filter_quirk set if we enable LFPS filter quirk

rx_detect_poll_quirk set if we enable rx_detect to polling lfps quirk

dis_u3_susphy_quirk set if we disable usb3 suspend phy

dis_u2_susphy_quirk set if we disable usb2 suspend phy

dis_enblslpm_quirk set if we clear enblslpm in GUSB2PHYCFG, disabling the suspend signal to the PHY.

dis_rxdet_inp3_quirk set if we disable Rx.Detect in P3

dis_u2_freeclk_exists_quirk set if we clear u2_freeclk_exists in GUSB2PHYCFG, specify that USB2 PHY doesn't provide a free-running PHY clock.

dis_del_phy_power_chg_quirk set if we disable delay phy power change quirk.

dis_tx_ipgap_linecheck_quirk set if we disable u2mac linestate check during HS transmit.

tx_de_emphasis_quirk set if we enable Tx de-emphasis quirk

tx_de_emphasis Tx de-emphasis value 0 - -6dB de-emphasis 1 - -3.5dB de-emphasis 2 - No de-emphasis 3 - Reserved

dis_metastability_quirk set to disable metastability quirk.

imod_interval set the interrupt moderation interval in 250ns increments or 0 to disable.

struct **dwc3_event_depevt**
Device Endpoint Events

Definition

```
struct dwc3_event_depevt {
  u32 one_bit:1;
  u32 endpoint_number:5;
  u32 endpoint_event:4;
  u32 reserved11_10:2;
  u32 status:4;
#define DEPEVT_STATUS_TRANSFER_ACTIVE   BIT(3);
#define DEPEVT_STATUS_BUSERR    BIT(0);
#define DEPEVT_STATUS_SHORT     BIT(1);
#define DEPEVT_STATUS_IOC       BIT(2);
#define DEPEVT_STATUS_LST       BIT(3);
#define DEPEVT_STREAMEVT_FOUND          1;
#define DEPEVT_STREAMEVT_NOTFOUND       2;
#define DEPEVT_STATUS_CONTROL_DATA      1;
#define DEPEVT_STATUS_CONTROL_STATUS    2;
#define DEPEVT_STATUS_CONTROL_PHASE(n)  ((n) & 3);
#define DEPEVT_TRANSFER_NO_RESOURCE     1;
#define DEPEVT_TRANSFER_BUS_EXPIRY      2;
  u32 parameters:16;
#define DEPEVT_PARAMETER_CMD(n) (((n) & (0xf << 8)) >> 8);
};
```

Members

one_bit indicates this is an endpoint event (not used)

endpoint_number number of the endpoint

endpoint_event The event we have: 0x00 - Reserved 0x01 - XferComplete 0x02 - XferInProgress 0x03 - XferNotReady 0x04 - RxTxFifoEvt (IN->Underrun, OUT->Overrun) 0x05 - Reserved 0x06 - StreamEvt 0x07 - EPCmdCmplt

reserved11_10 Reserved, don't use.

status Indicates the status of the event. Refer to databook for more information.

parameters Parameters of the current event. Refer to databook for more information.

struct **dwc3_event_devt**
Device Events

Definition

```
struct dwc3_event_devt {
  u32 one_bit:1;
  u32 device_event:7;
  u32 type:4;
  u32 reserved15_12:4;
  u32 event_info:9;
  u32 reserved31_25:7;
};
```

Members

one_bit indicates this is a non-endpoint event (not used)

device_event indicates it's a device event. Should read as 0x00

type indicates the type of device event. 0 - DisconnEvt 1 - USBRst 2 - ConnectDone 3 - ULStChng 4 - WkUpEvt 5 - Reserved 6 - EOPF 7 - SOF 8 - Reserved 9 - ErrticErr 10 - CmdCmplt 11 - EvntOverflow 12 - VndrDevTstRcved

reserved15_12 Reserved, not used

event_info Information about this event

reserved31_25 Reserved, not used

struct **dwc3_event_gevt**
Other Core Events

Definition

```
struct dwc3_event_gevt {
  u32 one_bit:1;
  u32 device_event:7;
  u32 phy_port_number:4;
  u32 reserved31_12:20;
};
```

Members

one_bit indicates this is a non-endpoint event (not used)

device_event indicates it's (0x03) Carkit or (0x04) I2C event.

phy_port_number self-explanatory

reserved31_12 Reserved, not used.

union **dwc3_event**
representation of Event Buffer contents

Definition

```
union dwc3_event {
  u32 raw;
  struct dwc3_event_type          type;
  struct dwc3_event_depevt        depevt;
  struct dwc3_event_devt          devt;
  struct dwc3_event_gevt          gevt;
};
```

Members

raw raw 32-bit event

type the type of the event

depevt Device Endpoint Event

devt Device Event

gevt Global Event

struct **dwc3_gadget_ep_cmd_params**
representation of endpoint command parameters

Definition

```
struct dwc3_gadget_ep_cmd_params {
  u32 param2;
  u32 param1;
  u32 param0;
};
```

Members

param2 third parameter

param1 second parameter

param0 first parameter

struct *dwc3_request* * **next_request**(struct list_head * *list*)
gets the next request on the given list

Parameters

struct list_head * list the request list to operate on

Description

Caller should take care of locking. This function return NULL or the first request available on **list**.

void **dwc3_gadget_move_started_request**(struct *dwc3_request* * *req*)
move **req** to the started_list

Parameters

struct dwc3_request * req the request to be moved

Description

Caller should take care of locking. This function will move **req** from its current list to the endpoint's started_list.

u32 **dwc3_gadget_ep_get_transfer_index**(struct *dwc3_ep* * *dep*)
Gets transfer index from HW

Parameters

struct dwc3_ep * dep dwc3 endpoint

Description

Caller should take care of locking. Returns the transfer resource index for a given endpoint.

int **dwc3_gadget_set_test_mode**(struct *dwc3* * *dwc*, int *mode*)
enables usb2 test modes

Parameters

struct dwc3 * dwc pointer to our context structure

int mode the mode to set (J, K SE0 NAK, Force Enable)

Description

Caller should take care of locking. This function will return 0 on success or -EINVAL if wrong Test Selector is passed.

int **dwc3_gadget_get_link_state**(struct *dwc3* * *dwc*)
gets current state of usb link

Parameters

struct dwc3 * dwc pointer to our context structure

Description

Caller should take care of locking. This function will return the link state on success (>= 0) or -ETIMEDOUT.

int **dwc3_gadget_set_link_state**(struct *dwc3* * *dwc*, enum dwc3_link_state *state*)
sets usb link to a particular state

Parameters

struct dwc3 * dwc pointer to our context structure

enum dwc3_link_state state the state to put link into

Description

Caller should take care of locking. This function will return 0 on success or -ETIMEDOUT.

void **dwc3_ep_inc_trb**(u8 * *index*)
increment a trb index.

Parameters

u8 * index Pointer to the TRB index to increment.

Description

The index should never point to the link TRB. After incrementing, if it is point to the link TRB, wrap around to the beginning. The link TRB is always at the last TRB entry.

void **dwc3_ep_inc_enq**(struct *dwc3_ep* * *dep*)
increment endpoint's enqueue pointer

Parameters

struct dwc3_ep * dep The endpoint whose enqueue pointer we're incrementing

void **dwc3_ep_inc_deq**(struct *dwc3_ep* * *dep*)
increment endpoint's dequeue pointer

Parameters

struct dwc3_ep * dep The endpoint whose enqueue pointer we're incrementing

void **dwc3_gadget_giveback**(struct *dwc3_ep* * *dep*, struct *dwc3_request* * *req*, int *status*)
call struct usb_request's ->complete callback

Parameters

struct dwc3_ep * dep The endpoint to whom the request belongs to

struct dwc3_request * req The request we're giving back

int status completion code for the request

Description

Must be called with controller's lock held and interrupts disabled. This function will unmap **req** and call its ->:c:func:*complete()* callback to notify upper layers that it has completed.

int **dwc3_send_gadget_generic_command**(struct *dwc3* * *dwc*, unsigned *cmd*, u32 *param*)
issue a generic command for the controller

Parameters

struct dwc3 * dwc pointer to the controller context

unsigned cmd the command to be issued

u32 param command parameter

Description

Caller should take care of locking. Issue **cmd** with a given **param** to **dwc** and wait for its completion.

int **dwc3_send_gadget_ep_cmd**(struct *dwc3_ep* * *dep*, unsigned *cmd*, struct *dwc3_gadget_ep_cmd_params* * *params*)
issue an endpoint command

Parameters

struct dwc3_ep * dep the endpoint to which the command is going to be issued

unsigned cmd the command to be issued

struct dwc3_gadget_ep_cmd_params * params parameters to the command

Description

Caller should handle locking. This function will issue **cmd** with given **params** to **dep** and wait for its completion.

int **dwc3_gadget_start_config**(struct *dwc3* * *dwc*, struct *dwc3_ep* * *dep*)
configure ep resources

Parameters

struct dwc3 * dwc pointer to our controller context structure

struct dwc3_ep * dep endpoint that is being enabled

Description

Issue a `DWC3_DEPCMD_DEPSTARTCFG` command to **dep**. After the command's completion, it will set Transfer Resource for all available endpoints.

The assignment of transfer resources cannot perfectly follow the data book due to the fact that the controller driver does not have all knowledge of the configuration in advance. It is given this information piecemeal by the composite gadget framework after every SET_CONFIGURATION and SET_INTERFACE. Trying to follow the databook programming model in this scenario can cause errors. For two reasons:

1) The databook says to do `DWC3_DEPCMD_DEPSTARTCFG` for every `USB_REQ_SET_CONFIGURATION` and `USB_REQ_SET_INTERFACE` (8.1.5). This is incorrect in the scenario of multiple interfaces.

2) The databook does not mention doing more `DWC3_DEPCMD_DEPXFERCFG` for new endpoint on alt setting (8.1.6).

The following simplified method is used instead:

All hardware endpoints can be assigned a transfer resource and this setting will stay persistent until either a core reset or hibernation. So whenever we do a `DWC3_DEPCMD_DEPSTARTCFG``(0) we can go ahead and do ``DWC3_DEPCMD_DEPXFERCFG` for every hardware endpoint as well. We are guaranteed that there are as many transfer resources as endpoints.

This function is called for each endpoint when it is being enabled but is triggered only when called for EP0-out, which always happens first, and which should only happen in one of the above conditions.

int **__dwc3_gadget_ep_enable**(struct *dwc3_ep* * *dep*, bool *modify*, bool *restore*)
initializes a hw endpoint

Parameters

struct dwc3_ep * dep endpoint to be initialized

bool modify if true, modify existing endpoint configuration

bool restore if true, restore endpoint configuration from scratch buffer

Description

Caller should take care of locking. Execute all necessary commands to initialize a HW endpoint so it can be used by a gadget driver.

int **__dwc3_gadget_ep_disable**(struct *dwc3_ep* * *dep*)
disables a hw endpoint

Parameters

struct dwc3_ep * dep the endpoint to disable

Description

This function undoes what __dwc3_gadget_ep_enable did and also removes requests which are currently being processed by the hardware and those which are not yet scheduled.

Caller should take care of locking.

void **dwc3_prepare_one_trb**(struct *dwc3_ep* * *dep*, struct *dwc3_request* * *req*, unsigned *chain*, unsigned *node*)
setup one TRB from one request

Parameters

struct dwc3_ep * dep endpoint for which this request is prepared

struct dwc3_request * req dwc3_request pointer

unsigned chain should this TRB be chained to the next?

unsigned node only for isochronous endpoints. First TRB needs different type.

struct *dwc3_trb* * **dwc3_ep_prev_trb**(struct *dwc3_ep* * *dep*, u8 *index*)
returns the previous TRB in the ring

Parameters

struct dwc3_ep * dep The endpoint with the TRB ring

u8 index The index of the current TRB in the ring

Description

Returns the TRB prior to the one pointed to by the index. If the index is 0, we will wrap backwards, skip the link TRB, and return the one just before that.

void **dwc3_gadget_setup_nump**(struct *dwc3* * *dwc*)
calculate and initialize NUMP field of `DWC3_DCFG`

Parameters

struct dwc3 * dwc pointer to our context structure

Description

The following looks like complex but it's actually very simple. In order to calculate the number of packets we can burst at once on OUT transfers, we're gonna use RxFIFO size.

To calculate RxFIFO size we need two numbers: MDWIDTH = size, in bits, of the internal memory bus RAM2_DEPTH = depth, in MDWIDTH, of internal RAM2 (where RxFIFO sits)

Given these two numbers, the formula is simple:

RxFIFO Size = (RAM2_DEPTH * MDWIDTH / 8) - 24 - 16;

24 bytes is for 3x SETUP packets 16 bytes is a clock domain crossing tolerance

Given RxFIFO Size, NUMP = RxFIFOSize / 1024;

int **dwc3_gadget_init**(struct *dwc3* * *dwc*)
initializes gadget related registers

Parameters

struct dwc3 * dwc pointer to our controller context structure

Description

Returns 0 on success otherwise negative errno.

DWC3_DEFAULT_AUTOSUSPEND_DELAY()
DesignWare USB3 DRD Controller Core file

Parameters

Description

Copyright (C) 2010-2011 Texas Instruments Incorporated - http://www.ti.com

Authors: Felipe Balbi <balbiti.com**>,** Sebastian Andrzej Siewior <bigeasy**linutronix.de**>

int **dwc3_get_dr_mode**(struct *dwc3* * *dwc*)
Validates and sets dr_mode

Parameters

struct dwc3 * dwc pointer to our context structure

int **dwc3_core_soft_reset**(struct *dwc3* * *dwc*)
Issues core soft reset and PHY reset

Parameters

struct dwc3 * dwc pointer to our context structure

void **dwc3_free_one_event_buffer**(struct *dwc3* * *dwc*, struct *dwc3_event_buffer* * *evt*)
Frees one event buffer

Parameters

struct dwc3 * dwc Pointer to our controller context structure

struct dwc3_event_buffer * evt Pointer to event buffer to be freed

struct *dwc3_event_buffer* * **dwc3_alloc_one_event_buffer**(struct *dwc3* * *dwc*, unsigned *length*)
Allocates one event buffer structure

Parameters

struct dwc3 * dwc Pointer to our controller context structure

unsigned length size of the event buffer

Description

Returns a pointer to the allocated event buffer structure on success otherwise ERR_PTR(errno).

void **dwc3_free_event_buffers**(struct *dwc3* * *dwc*)
frees all allocated event buffers

Parameters

struct dwc3 * dwc Pointer to our controller context structure

int **dwc3_alloc_event_buffers**(struct *dwc3* * *dwc*, unsigned *length*)
Allocates **num** event buffers of size **length**

Parameters

struct dwc3 * dwc pointer to our controller context structure

unsigned length size of event buffer

Description

Returns 0 on success otherwise negative errno. In the error case, dwc may contain some buffers allocated but not all which were requested.

int **dwc3_event_buffers_setup**(struct *dwc3* * *dwc*)
setup our allocated event buffers

Parameters

struct dwc3 * dwc pointer to our controller context structure

Description

Returns 0 on success otherwise negative errno.

int **dwc3_phy_setup**(struct *dwc3* * *dwc*)
Configure USB PHY Interface of DWC3 Core

Parameters

struct dwc3 * dwc Pointer to our controller context structure

Description

Returns 0 on success. The USB PHY interfaces are configured but not initialized. The PHY interfaces and the PHYs get initialized together with the core in dwc3_core_init.

int **dwc3_core_init**(struct *dwc3* * *dwc*)
Low-level initialization of DWC3 Core

Parameters

struct dwc3 * dwc Pointer to our controller context structure

Description

Returns 0 on success otherwise negative errno.

Writing a MUSB Glue Layer

Author Apelete Seketeli

Introduction

The Linux MUSB subsystem is part of the larger Linux USB subsystem. It provides support for embedded USB Device Controllers (UDC) that do not use Universal Host Controller Interface (UHCI) or Open Host Controller Interface (OHCI).

Instead, these embedded UDC rely on the USB On-the-Go (OTG) specification which they implement at least partially. The silicon reference design used in most cases is the Multipoint USB Highspeed Dual-Role Controller (MUSB HDRC) found in the Mentor Graphics Inventra™ design.

As a self-taught exercise I have written an MUSB glue layer for the Ingenic JZ4740 SoC, modelled after the many MUSB glue layers in the kernel source tree. This layer can be found at `drivers/usb/musb/jz4740.c`. In this documentation I will walk through the basics of the `jz4740.c` glue layer, explaining the different pieces and what needs to be done in order to write your own device glue layer.

Linux MUSB Basics

To get started on the topic, please read USB On-the-Go Basics (see Resources) which provides an introduction of USB OTG operation at the hardware level. A couple of wiki pages by Texas Instruments and Analog Devices also provide an overview of the Linux kernel MUSB configuration, albeit focused on some specific devices provided by these companies. Finally, getting acquainted with the USB specification at USB home page may come in handy, with practical instance provided through the Writing USB Device Drivers documentation (again, see Resources).

Linux USB stack is a layered architecture in which the MUSB controller hardware sits at the lowest. The MUSB controller driver abstract the MUSB controller hardware to the Linux USB stack:

```
  ------------------------
  |                      | <------- drivers/usb/gadget
  | Linux USB Core Stack | <------- drivers/usb/host
  |                      | <------- drivers/usb/core
  ------------------------
             ⬍
 --------------------------
 |                        | <------ drivers/usb/musb/musb_gadget.c
```

```
   | MUSB Controller driver | <------ drivers/usb/musb/musb_host.c
   |                        | <------ drivers/usb/musb/musb_core.c
   --------------------------
               ⬍
---------------------------------
| MUSB Platform Specific Driver |
|                               | <-- drivers/usb/musb/jz4740.c
|       aka "Glue Layer"        |
---------------------------------
               ⬍
---------------------------------
|   MUSB Controller Hardware    |
---------------------------------
```

As outlined above, the glue layer is actually the platform specific code sitting in between the controller driver and the controller hardware.

Just like a Linux USB driver needs to register itself with the Linux USB subsystem, the MUSB glue layer needs first to register itself with the MUSB controller driver. This will allow the controller driver to know about which device the glue layer supports and which functions to call when a supported device is detected or released; remember we are talking about an embedded controller chip here, so no insertion or removal at run-time.

All of this information is passed to the MUSB controller driver through a `platform_driver` structure defined in the glue layer as:

```
static struct platform_driver jz4740_driver = {
    .probe      = jz4740_probe,
    .remove     = jz4740_remove,
    .driver     = {
        .name   = "musb-jz4740",
    },
};
```

The probe and remove function pointers are called when a matching device is detected and, respectively, released. The name string describes the device supported by this glue layer. In the current case it matches a platform_device structure declared in `arch/mips/jz4740/platform.c`. Note that we are not using device tree bindings here.

In order to register itself to the controller driver, the glue layer goes through a few steps, basically allocating the controller hardware resources and initialising a couple of circuits. To do so, it needs to keep track of the information used throughout these steps. This is done by defining a private `jz4740_glue` structure:

```
struct jz4740_glue {
    struct device           *dev;
    struct platform_device  *musb;
    struct clk      *clk;
};
```

The dev and musb members are both device structure variables. The first one holds generic information about the device, since it's the basic device structure, and the latter holds information more closely related to the subsystem the device is registered to. The clk variable keeps information related to the device clock operation.

Let's go through the steps of the probe function that leads the glue layer to register itself to the controller driver.

Note:

For the sake of readability each function will be split in logical parts, each part being shown as if it was independent from the others.

```
static int jz4740_probe(struct platform_device *pdev)
{
    struct platform_device      *musb;
    struct jz4740_glue      *glue;
    struct clk                      *clk;
    int             ret;

    glue = devm_kzalloc(&pdev->dev, sizeof(*glue), GFP_KERNEL);
    if (!glue)
        return -ENOMEM;

    musb = platform_device_alloc("musb-hdrc", PLATFORM_DEVID_AUTO);
    if (!musb) {
        dev_err(&pdev->dev, "failed to allocate musb device\n");
        return -ENOMEM;
    }

    clk = devm_clk_get(&pdev->dev, "udc");
    if (IS_ERR(clk)) {
        dev_err(&pdev->dev, "failed to get clock\n");
        ret = PTR_ERR(clk);
        goto err_platform_device_put;
    }

    ret = clk_prepare_enable(clk);
    if (ret) {
        dev_err(&pdev->dev, "failed to enable clock\n");
        goto err_platform_device_put;
    }

    musb->dev.parent        = &pdev->dev;

    glue->dev           = &pdev->dev;
    glue->musb          = musb;
    glue->clk           = clk;

    return 0;

err_platform_device_put:
    platform_device_put(musb);
    return ret;
}
```

The first few lines of the probe function allocate and assign the glue, musb and clk variables. The GFP_KERNEL flag (line 8) allows the allocation process to sleep and wait for memory, thus being usable in a locking situation. The PLATFORM_DEVID_AUTO flag (line 12) allows automatic allocation and management of device IDs in order to avoid device namespace collisions with explicit IDs. With devm_clk_get() (line 18) the glue layer allocates the clock – the devm_ prefix indicates that clk_get() is managed: it automatically frees the allocated clock resource data when the device is released – and enable it.

Then comes the registration steps:

```
static int jz4740_probe(struct platform_device *pdev)
{
    struct musb_hdrc_platform_data  *pdata = &jz4740_musb_platform_data;

    pdata->platform_ops     = &jz4740_musb_ops;

    platform_set_drvdata(pdev, glue);

    ret = platform_device_add_resources(musb, pdev->resource,
                        pdev->num_resources);
    if (ret) {
        dev_err(&pdev->dev, "failed to add resources\n");
        goto err_clk_disable;
    }

    ret = platform_device_add_data(musb, pdata, sizeof(*pdata));
    if (ret) {
        dev_err(&pdev->dev, "failed to add platform_data\n");
        goto err_clk_disable;
    }

    return 0;

err_clk_disable:
    clk_disable_unprepare(clk);
err_platform_device_put:
    platform_device_put(musb);
    return ret;
}
```

The first step is to pass the device data privately held by the glue layer on to the controller driver through `platform_set_drvdata()` (line 7). Next is passing on the device resources information, also privately held at that point, through *`platform_device_add_resources()`* (line 9).

Finally comes passing on the platform specific data to the controller driver (line 16). Platform data will be discussed in *Device Platform Data*, but here we are looking at the `platform_ops` function pointer (line 5) in `musb_hdrc_platform_data` structure (line 3). This function pointer allows the MUSB controller driver to know which function to call for device operation:

```
static const struct musb_platform_ops jz4740_musb_ops = {
    .init       = jz4740_musb_init,
    .exit       = jz4740_musb_exit,
};
```

Here we have the minimal case where only init and exit functions are called by the controller driver when needed. Fact is the JZ4740 MUSB controller is a basic controller, lacking some features found in other controllers, otherwise we may also have pointers to a few other functions like a power management function or a function to switch between OTG and non-OTG modes, for instance.

At that point of the registration process, the controller driver actually calls the init function:

```
     static int jz4740_musb_init(struct musb *musb)
     {
         musb->xceiv = usb_get_phy(USB_PHY_TYPE_USB2);
         if (!musb->xceiv) {
             pr_err("HS UDC: no transceiver configured\n");
             return -ENODEV;
         }

         /* Silicon does not implement ConfigData register.
          * Set dyn_fifo to avoid reading EP config from hardware.
          */
         musb->dyn_fifo = true;

         musb->isr = jz4740_musb_interrupt;

         return 0;
     }
```

The goal of `jz4740_musb_init()` is to get hold of the transceiver driver data of the MUSB controller hardware and pass it on to the MUSB controller driver, as usual. The transceiver is the circuitry inside the controller hardware responsible for sending/receiving the USB data. Since it is an implementation of the physical layer of the OSI model, the transceiver is also referred to as PHY.

Getting hold of the MUSB PHY driver data is done with usb_get_phy() which returns a pointer to the structure containing the driver instance data. The next couple of instructions (line 12 and 14) are used as a quirk and to setup IRQ handling respectively. Quirks and IRQ handling will be discussed later in *Device Quirks* and *Handling IRQs*

```
static int jz4740_musb_exit(struct musb *musb)
{
    usb_put_phy(musb->xceiv);

    return 0;
}
```

Acting as the counterpart of init, the exit function releases the MUSB PHY driver when the controller hardware itself is about to be released.

Again, note that init and exit are fairly simple in this case due to the basic set of features of the JZ4740 controller hardware. When writing an musb glue layer for a more complex controller hardware, you might need to take care of more processing in those two functions.

Returning from the init function, the MUSB controller driver jumps back into the probe function:

```
static int jz4740_probe(struct platform_device *pdev)
{
    ret = platform_device_add(musb);
    if (ret) {
        dev_err(&pdev->dev, "failed to register musb device\n");
        goto err_clk_disable;
    }

    return 0;

err_clk_disable:
    clk_disable_unprepare(clk);
err_platform_device_put:
    platform_device_put(musb);
    return ret;
}
```

This is the last part of the device registration process where the glue layer adds the controller hardware device to Linux kernel device hierarchy: at this stage, all known information about the device is passed on to the Linux USB core stack:

```
    static int jz4740_remove(struct platform_device *pdev)
    {
        struct jz4740_glue  *glue = platform_get_drvdata(pdev);

        platform_device_unregister(glue->musb);
        clk_disable_unprepare(glue->clk);

        return 0;
    }
```

Acting as the counterpart of probe, the remove function unregister the MUSB controller hardware (line 5) and disable the clock (line 6), allowing it to be gated.

Handling IRQs

Additionally to the MUSB controller hardware basic setup and registration, the glue layer is also responsible for handling the IRQs:

```
    static irqreturn_t jz4740_musb_interrupt(int irq, void *__hci)
    {
        unsigned long   flags;
        irqreturn_t     retval = IRQ_NONE;
        struct musb     *musb = __hci;

        spin_lock_irqsave(&musb->lock, flags);

        musb->int_usb = musb_readb(musb->mregs, MUSB_INTRUSB);
        musb->int_tx = musb_readw(musb->mregs, MUSB_INTRTX);
        musb->int_rx = musb_readw(musb->mregs, MUSB_INTRRX);

        /*
         * The controller is gadget only, the state of the host mode IRQ bits is
         * undefined. Mask them to make sure that the musb driver core will
         * never see them set
         */
        musb->int_usb &= MUSB_INTR_SUSPEND | MUSB_INTR_RESUME |
            MUSB_INTR_RESET | MUSB_INTR_SOF;

        if (musb->int_usb || musb->int_tx || musb->int_rx)
            retval = musb_interrupt(musb);

        spin_unlock_irqrestore(&musb->lock, flags);

        return retval;
    }
```

Here the glue layer mostly has to read the relevant hardware registers and pass their values on to the controller driver which will handle the actual event that triggered the IRQ.

The interrupt handler critical section is protected by the `spin_lock_irqsave()` and counterpart `spin_unlock_irqrestore()` functions (line 7 and 24 respectively), which prevent the interrupt handler code to be run by two different threads at the same time.

Then the relevant interrupt registers are read (line 9 to 11):

- MUSB_INTRUSB: indicates which USB interrupts are currently active,
- MUSB_INTRTX: indicates which of the interrupts for TX endpoints are currently active,
- MUSB_INTRRX: indicates which of the interrupts for TX endpoints are currently active.

Note that `musb_readb()` is used to read 8-bit registers at most, while `musb_readw()` allows us to read at most 16-bit registers. There are other functions that can be used depending on the size of your device registers. See `musb_io.h` for more information.

Instruction on line 18 is another quirk specific to the JZ4740 USB device controller, which will be discussed later in *Device Quirks*.

The glue layer still needs to register the IRQ handler though. Remember the instruction on line 14 of the init function:

```
static int jz4740_musb_init(struct musb *musb)
{
    musb->isr = jz4740_musb_interrupt;

    return 0;
}
```

This instruction sets a pointer to the glue layer IRQ handler function, in order for the controller hardware to call the handler back when an IRQ comes from the controller hardware. The interrupt handler is now implemented and registered.

Device Platform Data

In order to write an MUSB glue layer, you need to have some data describing the hardware capabilities of your controller hardware, which is called the platform data.

Platform data is specific to your hardware, though it may cover a broad range of devices, and is generally found somewhere in the arch/ directory, depending on your device architecture.

For instance, platform data for the JZ4740 SoC is found in arch/mips/jz4740/platform.c. In the platform.c file each device of the JZ4740 SoC is described through a set of structures.

Here is the part of arch/mips/jz4740/platform.c that covers the USB Device Controller (UDC):

```
    /* USB Device Controller */
    struct platform_device jz4740_udc_xceiv_device = {
        .name = "usb_phy_gen_xceiv",
        .id   = 0,
    };

    static struct resource jz4740_udc_resources[] = {
        [0] = {
            .start = JZ4740_UDC_BASE_ADDR,
            .end   = JZ4740_UDC_BASE_ADDR + 0x10000 - 1,
            .flags = IORESOURCE_MEM,
        },
        [1] = {
            .start = JZ4740_IRQ_UDC,
            .end   = JZ4740_IRQ_UDC,
            .flags = IORESOURCE_IRQ,
            .name  = "mc",
        },
    };

    struct platform_device jz4740_udc_device = {
        .name = "musb-jz4740",
        .id   = -1,
        .dev  = {
            .dma_mask          = &jz4740_udc_device.dev.coherent_dma_mask,
            .coherent_dma_mask = DMA_BIT_MASK(32),
        },
        .num_resources = ARRAY_SIZE(jz4740_udc_resources),
        .resource      = jz4740_udc_resources,
    };
```

The jz4740_udc_xceiv_device platform device structure (line 2) describes the UDC transceiver with a name and id number.

At the time of this writing, note that usb_phy_gen_xceiv is the specific name to be used for all transceivers that are either built-in with reference USB IP or autonomous and doesn't require any PHY programming. You will need to set CONFIG_NOP_USB_XCEIV=y in the kernel configuration to make use of the corresponding transceiver driver. The id field could be set to -1 (equivalent to PLATFORM_DEVID_NONE), -2 (equivalent to PLATFORM_DEVID_AUTO) or start with 0 for the first device of this kind if we want a specific id number.

The jz4740_udc_resources resource structure (line 7) defines the UDC registers base addresses.

The first array (line 9 to 11) defines the UDC registers base memory addresses: start points to the first register memory address, end points to the last register memory address and the flags member defines the type of resource we are dealing with. So IORESOURCE_MEM is used to define the registers memory addresses. The second array (line 14 to 17) defines the UDC IRQ registers addresses. Since there is only one IRQ register available for the JZ4740 UDC, start and end point at the same address. The IORESOURCE_IRQ flag tells that we are dealing with IRQ resources, and the name mc is in fact hard-coded in the MUSB core in order for the controller driver to retrieve this IRQ resource by querying it by its name.

Finally, the jz4740_udc_device platform device structure (line 21) describes the UDC itself.

The musb-jz4740 name (line 22) defines the MUSB driver that is used for this device; remember this is in fact the name that we used in the jz4740_driver platform driver structure in *Linux MUSB Basics*. The id field (line 23) is set to -1 (equivalent to PLATFORM_DEVID_NONE) since we do not need an id for the device: the MUSB controller driver was already set to allocate an automatic id in *Linux MUSB Basics*. In the dev field we care for DMA related information here. The dma_mask field (line 25) defines the width of the DMA mask that is going to be used, and coherent_dma_mask (line 26) has the same purpose but for the alloc_coherent DMA mappings: in both cases we are using a 32 bits mask. Then the resource field (line 29) is simply a pointer to the resource structure defined before, while the num_resources field (line 28) keeps track of the number of arrays defined in the resource structure (in this case there were two resource arrays defined before).

With this quick overview of the UDC platform data at the arch/ level now done, let's get back to the MUSB glue layer specific platform data in drivers/usb/musb/jz4740.c:

```
static struct musb_hdrc_config jz4740_musb_config = {
    /* Silicon does not implement USB OTG. */
    .multipoint = 0,
    /* Max EPs scanned, driver will decide which EP can be used. */
    .num_eps    = 4,
    /* RAMbits needed to configure EPs from table */
    .ram_bits   = 9,
    .fifo_cfg = jz4740_musb_fifo_cfg,
    .fifo_cfg_size = ARRAY_SIZE(jz4740_musb_fifo_cfg),
};

static struct musb_hdrc_platform_data jz4740_musb_platform_data = {
    .mode   = MUSB_PERIPHERAL,
    .config = &jz4740_musb_config,
};
```

First the glue layer configures some aspects of the controller driver operation related to the controller hardware specifics. This is done through the jz4740_musb_config musb_hdrc_config structure.

Defining the OTG capability of the controller hardware, the multipoint member (line 3) is set to 0 (equivalent to false) since the JZ4740 UDC is not OTG compatible. Then num_eps (line 5) defines the number of USB endpoints of the controller hardware, including endpoint 0: here we have 3 endpoints + endpoint 0. Next is ram_bits (line 7) which is the width of the RAM address bus for the MUSB controller hardware. This information is needed when the controller driver cannot automatically configure endpoints by reading the relevant controller hardware registers. This issue will be discussed when we get to device quirks in *Device Quirks*. Last two fields (line 8 and 9) are also about device quirks: fifo_cfg points to the USB endpoints configuration table and fifo_cfg_size keeps track of the size of the number of entries in that configuration table. More on that later in *Device Quirks*.

Then this configuration is embedded inside jz4740_musb_platform_data musb_hdrc_platform_data structure (line 11): config is a pointer to the configuration structure itself, and mode tells the controller driver if the controller hardware may be used as MUSB_HOST only, MUSB_PERIPHERAL only or MUSB_OTG which is a dual mode.

Remember that jz4740_musb_platform_data is then used to convey platform data information as we have seen in the probe function in *Linux MUSB Basics*.

Device Quirks

Completing the platform data specific to your device, you may also need to write some code in the glue layer to work around some device specific limitations. These quirks may be due to some hardware bugs, or simply be the result of an incomplete implementation of the USB On-the-Go specification.

The JZ4740 UDC exhibits such quirks, some of which we will discuss here for the sake of insight even though these might not be found in the controller hardware you are working on.

Let's get back to the init function first:

```
    static int jz4740_musb_init(struct musb *musb)
    {
        musb->xceiv = usb_get_phy(USB_PHY_TYPE_USB2);
        if (!musb->xceiv) {
            pr_err("HS UDC: no transceiver configured\n");
            return -ENODEV;
        }

        /* Silicon does not implement ConfigData register.
         * Set dyn_fifo to avoid reading EP config from hardware.
         */
        musb->dyn_fifo = true;

        musb->isr = jz4740_musb_interrupt;

        return 0;
    }
```

Instruction on line 12 helps the MUSB controller driver to work around the fact that the controller hardware is missing registers that are used for USB endpoints configuration.

Without these registers, the controller driver is unable to read the endpoints configuration from the hardware, so we use line 12 instruction to bypass reading the configuration from silicon, and rely on a hard-coded table that describes the endpoints configuration instead:

```
static struct musb_fifo_cfg jz4740_musb_fifo_cfg[] = {
    { .hw_ep_num = 1, .style = FIFO_TX, .maxpacket = 512, },
    { .hw_ep_num = 1, .style = FIFO_RX, .maxpacket = 512, },
    { .hw_ep_num = 2, .style = FIFO_TX, .maxpacket = 64, },
};
```

Looking at the configuration table above, we see that each endpoints is described by three fields: hw_ep_num is the endpoint number, style is its direction (either FIFO_TX for the controller driver to send packets in the controller hardware, or FIFO_RX to receive packets from hardware), and maxpacket defines the maximum size of each data packet that can be transmitted over that endpoint. Reading from the table, the controller driver knows that endpoint 1 can be used to send and receive USB data packets of 512 bytes at once (this is in fact a bulk in/out endpoint), and endpoint 2 can be used to send data packets of 64 bytes at once (this is in fact an interrupt endpoint).

Note that there is no information about endpoint 0 here: that one is implemented by default in every silicon design, with a predefined configuration according to the USB specification. For more examples of endpoint configuration tables, see `musb_core.c`.

Let's now get back to the interrupt handler function:

```
    static irqreturn_t jz4740_musb_interrupt(int irq, void *__hci)
    {
        unsigned long   flags;
        irqreturn_t     retval = IRQ_NONE;
        struct musb     *musb = __hci;

        spin_lock_irqsave(&musb->lock, flags);

        musb->int_usb = musb_readb(musb->mregs, MUSB_INTRUSB);
        musb->int_tx = musb_readw(musb->mregs, MUSB_INTRTX);
        musb->int_rx = musb_readw(musb->mregs, MUSB_INTRRX);

        /*
         * The controller is gadget only, the state of the host mode IRQ bits is
         * undefined. Mask them to make sure that the musb driver core will
         * never see them set
         */
        musb->int_usb &= MUSB_INTR_SUSPEND | MUSB_INTR_RESUME |
            MUSB_INTR_RESET | MUSB_INTR_SOF;

        if (musb->int_usb || musb->int_tx || musb->int_rx)
            retval = musb_interrupt(musb);

        spin_unlock_irqrestore(&musb->lock, flags);

        return retval;
    }
```

Instruction on line 18 above is a way for the controller driver to work around the fact that some interrupt bits used for USB host mode operation are missing in the MUSB_INTRUSB register, thus left in an undefined hardware state, since this MUSB controller hardware is used in peripheral mode only. As a consequence, the glue layer masks these missing bits out to avoid parasite interrupts by doing a logical AND operation between the value read from MUSB_INTRUSB and the bits that are actually implemented in the register.

These are only a couple of the quirks found in the JZ4740 USB device controller. Some others were directly addressed in the MUSB core since the fixes were generic enough to provide a better handling of the issues for others controller hardware eventually.

Conclusion

Writing a Linux MUSB glue layer should be a more accessible task, as this documentation tries to show the ins and outs of this exercise.

The JZ4740 USB device controller being fairly simple, I hope its glue layer serves as a good example for the curious mind. Used with the current MUSB glue layers, this documentation should provide enough guidance to get started; should anything gets out of hand, the linux-usb mailing list archive is another helpful resource to browse through.

Acknowledgements

Many thanks to Lars-Peter Clausen and Maarten ter Huurne for answering my questions while I was writing the JZ4740 glue layer and for helping me out getting the code in good shape.

I would also like to thank the Qi-Hardware community at large for its cheerful guidance and support.

Resources

USB Home Page: http://www.usb.org

linux-usb Mailing List Archives: http://marc.info/?l=linux-usb

USB On-the-Go Basics: http://www.maximintegrated.com/app-notes/index.mvp/id/1822

Writing USB Device Drivers

Texas Instruments USB Configuration Wiki Page: http://processors.wiki.ti.com/index.php/Usbgeneralpage

Analog Devices Blackfin MUSB Configuration: http://docs.blackfin.uclinux.org/doku.php?id=linux-kernel:drivers:musb

USB Type-C connector class

Introduction

The typec class is meant for describing the USB Type-C ports in a system to the user space in unified fashion. The class is designed to provide nothing else except the user space interface implementation in hope that it can be utilized on as many platforms as possible.

The platforms are expected to register every USB Type-C port they have with the class. In a normal case the registration will be done by a USB Type-C or PD PHY driver, but it may be a driver for firmware interface such as UCSI, driver for USB PD controller or even driver for Thunderbolt3 controller. This document considers the component registering the USB Type-C ports with the class as "port driver".

On top of showing the capabilities, the class also offer user space control over the roles and alternate modes of ports, partners and cable plugs when the port driver is capable of supporting those features.

The class provides an API for the port drivers described in this document. The attributes are described in Documentation/ABI/testing/sysfs-class-typec.

User space interface

Every port will be presented as its own device under /sys/class/typec/. The first port will be named "port0", the second "port1" and so on.

When connected, the partner will be presented also as its own device under /sys/class/typec/. The parent of the partner device will always be the port it is attached to. The partner attached to port "port0" will be named "port0-partner". Full path to the device would be /sys/class/typec/port0/port0-partner/.

The cable and the two plugs on it may also be optionally presented as their own devices under /sys/class/typec/. The cable attached to the port "port0" port will be named port0-cable and the plug on the SOP Prime end (see USB Power Delivery Specification ch. 2.4) will be named "port0-plug0" and on the SOP Double Prime end "port0-plug1". The parent of a cable will always be the port, and the parent of the cable plugs will always be the cable.

If the port, partner or cable plug supports Alternate Modes, every supported Alternate Mode SVID will have their own device describing them. Note that the Alternate Mode devices will not be attached to the typec class. The parent of an alternate mode will be the device that supports it, so for example an alternate mode of port0-partner will be presented under /sys/class/typec/port0-partner/. Every mode that is supported will have its own group under the Alternate Mode device named "mode<index>", for example /sys/class/typec/port0/<alternate mode>/mode1/. The requests for entering/exiting a mode can be done with "active" attribute file in that group.

Driver API

Registering the ports

The port drivers will describe every Type-C port they control with struct typec_capability data structure, and register them with the following API:

struct typec_port * **typec_register_port**(struct *device* * *parent*, const struct typec_capability * *cap*)
Register a USB Type-C Port

Parameters

struct device * parent Parent device

const struct typec_capability * cap Description of the port

Description

Registers a device for USB Type-C Port described in **cap**.

Returns handle to the port on success or NULL on failure.

void **typec_unregister_port**(struct typec_port * *port*)
Unregister a USB Type-C Port

Parameters

`struct typec_port * port` The port to be unregistered

Description

Unregister device created with *`typec_register_port()`*.

When registering the ports, the prefer_role member in struct typec_capability deserves special notice. If the port that is being registered does not have initial role preference, which means the port does not execute Try.SNK or Try.SRC by default, the member must have value TYPEC_NO_PREFERRED_ROLE. Otherwise if the port executes Try.SNK by default, the member must have value TYPEC_DEVICE, and with Try.SRC the value must be TYPEC_HOST.

Registering Partners

After successful connection of a partner, the port driver needs to register the partner with the class. Details about the partner need to be described in struct typec_partner_desc. The class copies the details of the partner during registration. The class offers the following API for registering/unregistering partners.

struct typec_partner * **`typec_register_partner`**(struct typec_port * *port*, struct typec_partner_desc * *desc*)
Register a USB Type-C Partner

Parameters

`struct typec_port * port` The USB Type-C Port the partner is connected to

`struct typec_partner_desc * desc` Description of the partner

Description

Registers a device for USB Type-C Partner described in **desc**.

Returns handle to the partner on success or NULL on failure.

void **`typec_unregister_partner`**(struct typec_partner * *partner*)
Unregister a USB Type-C Partner

Parameters

`struct typec_partner * partner` The partner to be unregistered

Description

Unregister device created with *`typec_register_partner()`*.

The class will provide a handle to struct typec_partner if the registration was successful, or NULL.

If the partner is USB Power Delivery capable, and the port driver is able to show the result of Discover Identity command, the partner descriptor structure should include handle to struct usb_pd_identity instance. The class will then create a sysfs directory for the identity under the partner device. The result of Discover Identity command can then be reported with the following API:

int **`typec_partner_set_identity`**(struct typec_partner * *partner*)
Report result from Discover Identity command

Parameters

`struct typec_partner * partner` The partner updated identity values

Description

This routine is used to report that the result of Discover Identity USB power delivery command has become available.

Registering Cables

After successful connection of a cable that supports USB Power Delivery Structured VDM "Discover Identity", the port driver needs to register the cable and one or two plugs, depending if there is CC Double Prime controller present in the cable or not. So a cable capable of SOP Prime communication, but not SOP Double Prime communication, should only have one plug registered. For more information about SOP communication, please read chapter about it from the latest USB Power Delivery specification.

The plugs are represented as their own devices. The cable is registered first, followed by registration of the cable plugs. The cable will be the parent device for the plugs. Details about the cable need to be described in struct typec_cable_desc and about a plug in struct typec_plug_desc. The class copies the details during registration. The class offers the following API for registering/unregistering cables and their plugs:

struct typec_plug * **typec_register_plug**(struct typec_cable * *cable*, struct typec_plug_desc * *desc*)
Register a USB Type-C Cable Plug

Parameters

struct typec_cable * cable USB Type-C Cable with the plug

struct typec_plug_desc * desc Description of the cable plug

Description

Registers a device for USB Type-C Cable Plug described in **desc**. A USB Type-C Cable Plug represents a plug with electronics in it that can response to USB Power Delivery SOP Prime or SOP Double Prime packages.

Returns handle to the cable plug on success or NULL on failure.

void **typec_unregister_plug**(struct typec_plug * *plug*)
Unregister a USB Type-C Cable Plug

Parameters

struct typec_plug * plug The cable plug to be unregistered

Description

Unregister device created with *typec_register_plug()*.

struct typec_cable * **typec_register_cable**(struct typec_port * *port*, struct typec_cable_desc * *desc*)
Register a USB Type-C Cable

Parameters

struct typec_port * port The USB Type-C Port the cable is connected to

struct typec_cable_desc * desc Description of the cable

Description

Registers a device for USB Type-C Cable described in **desc**. The cable will be parent for the optional cable plug devises.

Returns handle to the cable on success or NULL on failure.

void **typec_unregister_cable**(struct typec_cable * *cable*)
Unregister a USB Type-C Cable

Parameters

struct typec_cable * cable The cable to be unregistered

Description

Unregister device created with *typec_register_cable()*.

The class will provide a handle to struct typec_cable and struct typec_plug if the registration is successful, or NULL if it isn't.

If the cable is USB Power Delivery capable, and the port driver is able to show the result of Discover Identity command, the cable descriptor structure should include handle to struct usb_pd_identity instance. The class will then create a sysfs directory for the identity under the cable device. The result of Discover Identity command can then be reported with the following API:

int **typec_cable_set_identity**(struct typec_cable * *cable*)
    Report result from Discover Identity command

Parameters

struct typec_cable * cable The cable updated identity values

Description

This routine is used to report that the result of Discover Identity USB power delivery command has become available.

Notifications

When the partner has executed a role change, or when the default roles change during connection of a partner or cable, the port driver must use the following APIs to report it to the class:

void **typec_set_data_role**(struct typec_port * *port*, enum typec_data_role *role*)
    Report data role change

Parameters

struct typec_port * port The USB Type-C Port where the role was changed

enum typec_data_role role The new data role

Description

This routine is used by the port drivers to report data role changes.

void **typec_set_pwr_role**(struct typec_port * *port*, enum typec_role *role*)
    Report power role change

Parameters

struct typec_port * port The USB Type-C Port where the role was changed

enum typec_role role The new data role

Description

This routine is used by the port drivers to report power role changes.

void **typec_set_vconn_role**(struct typec_port * *port*, enum typec_role *role*)
    Report VCONN source change

Parameters

struct typec_port * port The USB Type-C Port which VCONN role changed

enum typec_role role Source when **port** is sourcing VCONN, or Sink when it's not

Description

This routine is used by the port drivers to report if the VCONN source is changes.

void **typec_set_pwr_opmode**(struct typec_port * *port*, enum typec_pwr_opmode *opmode*)
    Report changed power operation mode

Parameters

struct typec_port * port The USB Type-C Port where the mode was changed

enum typec_pwr_opmode opmode New power operation mode

Description

This routine is used by the port drivers to report changed power operation mode in **port**. The modes are USB (default), 1.5A, 3.0A as defined in USB Type-C specification, and "USB Power Delivery" when the power levels are negotiated with methods defined in USB Power Delivery specification.

Alternate Modes

USB Type-C ports, partners and cable plugs may support Alternate Modes. Each Alternate Mode will have identifier called SVID, which is either a Standard ID given by USB-IF or vendor ID, and each supported SVID can have 1 - 6 modes. The class provides struct typec_mode_desc for describing individual mode of a SVID, and struct typec_altmode_desc which is a container for all the supported modes.

Ports that support Alternate Modes need to register each SVID they support with the following API:

struct typec_altmode * **typec_port_register_altmode**(struct typec_port * *port*, const struct typec_altmode_desc * *desc*)

Register USB Type-C Port Alternate Mode

Parameters

struct typec_port * port USB Type-C Port that supports the alternate mode

const struct typec_altmode_desc * desc Description of the alternate mode

Description

This routine is used to register an alternate mode that **port** is capable of supporting.

Returns handle to the alternate mode on success or NULL on failure.

If a partner or cable plug provides a list of SVIDs as response to USB Power Delivery Structured VDM Discover SVIDs message, each SVID needs to be registered.

API for the partners:

struct typec_altmode * **typec_partner_register_altmode**(struct typec_partner * *partner*, const struct typec_altmode_desc * *desc*)

Register USB Type-C Partner Alternate Mode

Parameters

struct typec_partner * partner USB Type-C Partner that supports the alternate mode

const struct typec_altmode_desc * desc Description of the alternate mode

Description

This routine is used to register each alternate mode individually that **partner** has listed in response to Discover SVIDs command. The modes for a SVID listed in response to Discover Modes command need to be listed in an array in **desc**.

Returns handle to the alternate mode on success or NULL on failure.

API for the Cable Plugs:

struct typec_altmode * **typec_plug_register_altmode**(struct typec_plug * *plug*, const struct typec_altmode_desc * *desc*)

Register USB Type-C Cable Plug Alternate Mode

Parameters

struct typec_plug * plug USB Type-C Cable Plug that supports the alternate mode

const struct typec_altmode_desc * desc Description of the alternate mode

Description

This routine is used to register each alternate mode individually that **plug** has listed in response to Discover SVIDs command. The modes for a SVID that the plug lists in response to Discover Modes command need to be listed in an array in **desc**.

Returns handle to the alternate mode on success or NULL on failure.

So ports, partners and cable plugs will register the alternate modes with their own functions, but the registration will always return a handle to struct typec_altmode on success, or NULL. The unregistration will happen with the same function:

void **typec_unregister_altmode**(struct typec_altmode * *alt*)
    Unregister Alternate Mode

Parameters

struct typec_altmode * alt The alternate mode to be unregistered

Description

Unregister device created with *typec_partner_register_altmode()*, *typec_plug_register_altmode()* or *typec_port_register_altmode()*.

If a partner or cable plug enters or exits a mode, the port driver needs to notify the class with the following API:

void **typec_altmode_update_active**(struct typec_altmode * *alt*, int *mode*, bool *active*)
    Report Enter/Exit mode

Parameters

struct typec_altmode * alt Handle to the alternate mode

int mode Mode index

bool active True when the mode has been entered

Description

If a partner or cable plug executes Enter/Exit Mode command successfully, the drivers use this routine to report the updated state of the mode.

USB3 debug port

Author Lu Baolu <baolu.lu@linux.intel.com>

Date March 2017

GENERAL

This is a HOWTO for using the USB3 debug port on x86 systems.

Before using any kernel debugging functionality based on USB3 debug port, you need to:

```
1) check whether any USB3 debug port is available in
   your system;
2) check which port is used for debugging purposes;
3) have a USB 3.0 super-speed A-to-A debugging cable.
```

INTRODUCTION

The xHCI debug capability (DbC) is an optional but standalone functionality provided by the xHCI host controller. The xHCI specification describes DbC in the section 7.6.

When DbC is initialized and enabled, it will present a debug device through the debug port (normally the first USB3 super-speed port). The debug device is fully compliant with the USB framework and provides the equivalent of a very high performance full-duplex serial link between the debug target (the system under debugging) and a debug host.

EARLY PRINTK

DbC has been designed to log early printk messages. One use for this feature is kernel debugging. For example, when your machine crashes very early before the regular console code is initialized. Other uses include simpler, lockless logging instead of a full- blown printk console driver and klogd.

On the debug target system, you need to customize a debugging kernel with CONFIG_EARLY_PRINTK_USB_XDBC enabled. And, add below kernel boot parameter:

```
"earlyprintk=xdbc"
```

If there are multiple xHCI controllers in your system, you can append a host contoller index to this kernel parameter. This index starts from 0.

Current design doesn't support DbC runtime suspend/resume. As the result, you'd better disable runtime power management for USB subsystem by adding below kernel boot parameter:

```
"usbcore.autosuspend=-1"
```

Before starting the debug target, you should connect the debug port to a USB port (root port or port of any external hub) on the debug host. The cable used to connect these two ports should be a USB 3.0 super-speed A-to-A debugging cable.

During early boot of the debug target, DbC will be detected and initialized. After initialization, the debug host should be able to enumerate the debug device in debug target. The debug host will then bind the debug device with the usb_debug driver module and create the /dev/ttyUSB device.

If the debug device enumeration goes smoothly, you should be able to see below kernel messages on the debug host:

```
# tail -f /var/log/kern.log
[ 1815.983374] usb 4-3: new SuperSpeed USB device number 4 using xhci_hcd
[ 1815.999595] usb 4-3: LPM exit latency is zeroed, disabling LPM.
[ 1815.999899] usb 4-3: New USB device found, idVendor=1d6b, idProduct=0004
[ 1815.999902] usb 4-3: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[ 1815.999903] usb 4-3: Product: Remote GDB
[ 1815.999904] usb 4-3: Manufacturer: Linux
[ 1815.999905] usb 4-3: SerialNumber: 0001
[ 1816.000240] usb_debug 4-3:1.0: xhci_dbc converter detected
[ 1816.000360] usb 4-3: xhci_dbc converter now attached to ttyUSB0
```

You can use any communication program, for example minicom, to read and view the messages. Below simple bash scripts can help you to check the sanity of the setup.

```
===== start of bash scripts =============
#!/bin/bash

while true ; do
        while [ ! -d /sys/class/tty/ttyUSB0 ] ; do
                :
        done
cat /dev/ttyUSB0
```

```
done
===== end of bash scripts ===============
```

Serial TTY

The DbC support has been added to the xHCI driver. You can get a debug device provided by the DbC at runtime.

In order to use this, you need to make sure your kernel has been configured to support USB_XHCI_DBGCAP. A sysfs attribute under the xHCI device node is used to enable or disable DbC. By default, DbC is disabled:

```
root@target:/sys/bus/pci/devices/0000:00:14.0# cat dbc
disabled
```

Enable DbC with the following command:

```
root@target:/sys/bus/pci/devices/0000:00:14.0# echo enable > dbc
```

You can check the DbC state at anytime:

```
root@target:/sys/bus/pci/devices/0000:00:14.0# cat dbc
enabled
```

Connect the debug target to the debug host with a USB 3.0 super- speed A-to-A debugging cable. You can see /dev/ttyDBC0 created on the debug target. You will see below kernel message lines:

```
root@target: tail -f /var/log/kern.log
[  182.730103] xhci_hcd 0000:00:14.0: DbC connected
[  191.169420] xhci_hcd 0000:00:14.0: DbC configured
[  191.169597] xhci_hcd 0000:00:14.0: DbC now attached to /dev/ttyDBC0
```

Accordingly, the DbC state has been brought up to:

```
root@target:/sys/bus/pci/devices/0000:00:14.0# cat dbc
configured
```

On the debug host, you will see the debug device has been enumerated. You will see below kernel message lines:

```
root@host: tail -f /var/log/kern.log
[   79.454780] usb 2-2.1: new SuperSpeed USB device number 3 using xhci_hcd
[   79.475003] usb 2-2.1: LPM exit latency is zeroed, disabling LPM.
[   79.475389] usb 2-2.1: New USB device found, idVendor=1d6b, idProduct=0010
[   79.475390] usb 2-2.1: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[   79.475391] usb 2-2.1: Product: Linux USB Debug Target
[   79.475392] usb 2-2.1: Manufacturer: Linux Foundation
[   79.475393] usb 2-2.1: SerialNumber: 0001
```

The debug device works now. You can use any communication or debugging program to talk between the host and the target.

CHAPTER

FOURTEEN

PCI SUPPORT LIBRARY

unsigned char **pci_bus_max_busnr**(struct pci_bus * *bus*)
returns maximum PCI bus number of given bus' children

Parameters

struct pci_bus * bus pointer to PCI bus structure to search

Description

Given a PCI bus, returns the highest PCI bus number present in the set including the given PCI bus and its list of child PCI buses.

int **pci_find_capability**(struct pci_dev * *dev*, int *cap*)
query for devices' capabilities

Parameters

struct pci_dev * dev PCI device to query

int cap capability code

Description

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

PCI_CAP_ID_PM Power Management PCI_CAP_ID_AGP Accelerated Graphics Port PCI_CAP_ID_VPD Vital Product Data PCI_CAP_ID_SLOTID Slot Identification PCI_CAP_ID_MSI Message Signalled Interrupts PCI_CAP_ID_CHSWP CompactPCI HotSwap PCI_CAP_ID_PCIX PCI-X PCI_CAP_ID_EXP PCI Express

int **pci_bus_find_capability**(struct pci_bus * *bus*, unsigned int *devfn*, int *cap*)
query for devices' capabilities

Parameters

struct pci_bus * bus the PCI bus to query

unsigned int devfn PCI device to query

int cap capability code

Description

Like *pci_find_capability()* but works for pci devices that do not have a pci_dev structure set up yet.

Returns the address of the requested capability structure within the device's PCI configuration space or 0 in case the device does not support it.

int **pci_find_next_ext_capability**(struct pci_dev * *dev*, int *start*, int *cap*)
Find an extended capability

Parameters

struct pci_dev * dev PCI device to query

`int start` address at which to start looking (0 to start at beginning of list)

`int cap` capability code

Description

Returns the address of the next matching extended capability structure within the device's PCI configuration space or 0 if the device does not support it. Some capabilities can occur several times, e.g., the vendor-specific capability, and this provides a way to find them all.

int **`pci_find_ext_capability`**(struct pci_dev * *dev*, int *cap*)
    Find an extended capability

Parameters

`struct pci_dev * dev` PCI device to query

`int cap` capability code

Description

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

> PCI_EXT_CAP_ID_ERR Advanced Error Reporting PCI_EXT_CAP_ID_VC Virtual Channel PCI_EXT_CAP_ID_DSN Device Serial Number PCI_EXT_CAP_ID_PWR Power Budgeting

int **`pci_find_next_ht_capability`**(struct pci_dev * *dev*, int *pos*, int *ht_cap*)
    query a device's Hypertransport capabilities

Parameters

`struct pci_dev * dev` PCI device to query

`int pos` Position from which to continue searching

`int ht_cap` Hypertransport capability code

Description

To be used in conjunction with *`pci_find_ht_capability()`* to search for all capabilities matching **ht_cap**. **pos** should always be a value returned from *`pci_find_ht_capability()`*.

NB. To be 100% safe against broken PCI devices, the caller should take steps to avoid an infinite loop.

int **`pci_find_ht_capability`**(struct pci_dev * *dev*, int *ht_cap*)
    query a device's Hypertransport capabilities

Parameters

`struct pci_dev * dev` PCI device to query

`int ht_cap` Hypertransport capability code

Description

Tell if a device supports a given Hypertransport capability. Returns an address within the device's PCI configuration space or 0 in case the device does not support the request capability. The address points to the PCI capability, of type PCI_CAP_ID_HT, which has a Hypertransport capability matching **ht_cap**.

struct resource * **`pci_find_parent_resource`**(const struct pci_dev * *dev*, struct resource * *res*)
    return resource region of parent bus of given region

Parameters

`const struct pci_dev * dev` PCI device structure contains resources to be searched

`struct resource * res` child resource record for which parent is sought

Description

> For given resource region of given device, return the resource region of parent bus the given region is contained in.

struct resource * **pci_find_resource**(struct pci_dev * *dev*, struct resource * *res*)
Return matching PCI device resource

Parameters

struct pci_dev * dev PCI device to query

struct resource * res Resource to look for

Description

Goes over standard PCI resources (BARs) and checks if the given resource is partially or fully contained in any of them. In that case the matching resource is returned, NULL otherwise.

struct pci_dev * **pci_find_pcie_root_port**(struct pci_dev * *dev*)
return PCIe Root Port

Parameters

struct pci_dev * dev PCI device to query

Description

Traverse up the parent chain and return the PCIe Root Port PCI Device for a given PCI Device.

int **__pci_complete_power_transition**(struct pci_dev * *dev*, pci_power_t *state*)
Complete power transition of a PCI device

Parameters

struct pci_dev * dev PCI device to handle.

pci_power_t state State to put the device into.

Description

This function should not be called directly by device drivers.

int **pci_set_power_state**(struct pci_dev * *dev*, pci_power_t *state*)
Set the power state of a PCI device

Parameters

struct pci_dev * dev PCI device to handle.

pci_power_t state PCI power state (D0, D1, D2, D3hot) to put the device into.

Description

Transition a device to a new power state, using the platform firmware and/or the device's PCI PM registers.

RETURN VALUE: -EINVAL if the requested state is invalid. -EIO if device does not support PCI PM or its PM capabilities register has a wrong version, or device doesn't support the requested state. 0 if the transition is to D1 or D2 but D1 and D2 are not supported. 0 if device already is in the requested state. 0 if the transition is to D3 but D3 is not supported. 0 if device's power state has been successfully changed.

pci_power_t **pci_choose_state**(struct pci_dev * *dev*, pm_message_t *state*)
Choose the power state of a PCI device

Parameters

struct pci_dev * dev PCI device to be suspended

pm_message_t state target sleep state for the whole system. This is the value that is passed to suspend() function.

Description

Returns PCI power state suitable for given device and given system message.

int **pci_save_state**(struct pci_dev * *dev*)
save the PCI configuration space of a device before suspending

Parameters

struct pci_dev * dev

- PCI device that we're dealing with

void **pci_restore_state**(struct pci_dev * *dev*)
Restore the saved state of a PCI device

Parameters

struct pci_dev * dev

- PCI device that we're dealing with

struct pci_saved_state * **pci_store_saved_state**(struct pci_dev * *dev*)
Allocate and return an opaque struct containing the device saved state.

Parameters

struct pci_dev * dev PCI device that we're dealing with

Description

Return NULL if no state or error.

int **pci_load_saved_state**(struct pci_dev * *dev*, struct pci_saved_state * *state*)
Reload the provided save state into struct pci_dev.

Parameters

struct pci_dev * dev PCI device that we're dealing with

struct pci_saved_state * state Saved state returned from *pci_store_saved_state()*

int **pci_load_and_free_saved_state**(struct pci_dev * *dev*, struct pci_saved_state ** *state*)
Reload the save state pointed to by state, and free the memory allocated for it.

Parameters

struct pci_dev * dev PCI device that we're dealing with

struct pci_saved_state ** state Pointer to saved state returned from *pci_store_saved_state()*

int **pci_reenable_device**(struct pci_dev * *dev*)
Resume abandoned device

Parameters

struct pci_dev * dev PCI device to be resumed

Description

Note this function is a backend of pci_default_resume and is not supposed to be called by normal code, write proper resume handler and use it instead.

int **pci_enable_device_io**(struct pci_dev * *dev*)
Initialize a device for use with IO space

Parameters

struct pci_dev * dev PCI device to be initialized

Description

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

int **pci_enable_device_mem**(struct pci_dev * *dev*)
Initialize a device for use with Memory space

Parameters

struct pci_dev * dev PCI device to be initialized

Description

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

int **pci_enable_device**(struct pci_dev * *dev*)
Initialize device before it's used by a driver.

Parameters

struct pci_dev * dev PCI device to be initialized

Description

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

Note we don't actually enable the device many times if we call this function repeatedly (we just increment the count).

int **pcim_enable_device**(struct pci_dev * *pdev*)
Managed *pci_enable_device()*

Parameters

struct pci_dev * pdev PCI device to be initialized

Description

Managed *pci_enable_device()*.

void **pcim_pin_device**(struct pci_dev * *pdev*)
Pin managed PCI device

Parameters

struct pci_dev * pdev PCI device to pin

Description

Pin managed PCI device **pdev**. Pinned device won't be disabled on driver detach. **pdev** must have been enabled with *pcim_enable_device()*.

void **pci_disable_device**(struct pci_dev * *dev*)
Disable PCI device after use

Parameters

struct pci_dev * dev PCI device to be disabled

Description

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

Note we don't actually disable the device until all callers of *pci_enable_device()* have called *pci_disable_device()*.

int **pci_set_pcie_reset_state**(struct pci_dev * *dev*, enum pcie_reset_state *state*)
set reset state for device dev

Parameters

struct pci_dev * dev the PCIe device reset

enum pcie_reset_state state Reset state to enter into

Description

Sets the PCI reset state for the device.

bool **pci_pme_capable**(struct pci_dev * *dev*, pci_power_t *state*)
check the capability of PCI device to generate PME#

Parameters

`struct pci_dev * dev` PCI device to handle.

`pci_power_t state` PCI state from which device will issue PME#.

void **`pci_pme_active`**(struct pci_dev * *dev*, bool *enable*)
enable or disable PCI device's PME# function

Parameters

`struct pci_dev * dev` PCI device to handle.

`bool enable` 'true' to enable PME# generation; 'false' to disable it.

Description

The caller must verify that the device is capable of generating PME# before calling this function with **enable** equal to 'true'.

int **`pci_enable_wake`**(struct pci_dev * *dev*, pci_power_t *state*, bool *enable*)
enable PCI device as wakeup event source

Parameters

`struct pci_dev * dev` PCI device affected

`pci_power_t state` PCI state from which device will issue wakeup events

`bool enable` True to enable event generation; false to disable

Description

This enables the device as a wakeup event source, or disables it. When such events involves platform-specific hooks, those hooks are called automatically by this routine.

Devices with legacy power management (no standard PCI PM capabilities) always require such platform hooks.

RETURN VALUE: 0 is returned on success -EINVAL is returned if device is not supposed to wake up the system Error code depending on the platform is returned if both the platform and the native mechanism fail to enable the generation of wake-up events

int **`pci_wake_from_d3`**(struct pci_dev * *dev*, bool *enable*)
enable/disable device to wake up from D3_hot or D3_cold

Parameters

`struct pci_dev * dev` PCI device to prepare

`bool enable` True to enable wake-up event generation; false to disable

Description

Many drivers want the device to wake up the system from D3_hot or D3_cold and this function allows them to set that up cleanly - *`pci_enable_wake()`* should not be called twice in a row to enable wake-up due to PCI PM vs ACPI ordering constraints.

This function only returns error code if the device is not capable of generating PME# from both D3_hot and D3_cold, and the platform is unable to enable wake-up power for it.

int **`pci_prepare_to_sleep`**(struct pci_dev * *dev*)
prepare PCI device for system-wide transition into a sleep state

Parameters

`struct pci_dev * dev` Device to handle.

Description

Choose the power state appropriate for the device depending on whether it can wake up the system and/or is power manageable by the platform (PCI_D3hot is the default) and put the device into that state.

int **`pci_back_from_sleep`**(struct pci_dev * *dev*)
turn PCI device on during system-wide transition into working state

Parameters

struct pci_dev * dev Device to handle.

Description

Disable device's system wake-up capability and put it into D0.

bool **pci_dev_run_wake**(struct pci_dev * *dev*)
Check if device can generate run-time wake-up events.

Parameters

struct pci_dev * dev Device to check.

Description

Return true if the device itself is capable of generating wake-up events (through the platform or using the native PCIe PME) or if the device supports PME and one of its upstream bridges can generate wake-up events.

void **pci_d3cold_enable**(struct pci_dev * *dev*)
Enable D3cold for device

Parameters

struct pci_dev * dev PCI device to handle

Description

This function can be used in drivers to enable D3cold from the device they handle. It also updates upstream PCI bridge PM capabilities accordingly.

void **pci_d3cold_disable**(struct pci_dev * *dev*)
Disable D3cold for device

Parameters

struct pci_dev * dev PCI device to handle

Description

This function can be used in drivers to disable D3cold from the device they handle. It also updates upstream PCI bridge PM capabilities accordingly.

int **pci_enable_atomic_ops_to_root**(struct pci_dev * *dev*, u32 *cap_mask*)
enable AtomicOp requests to root port

Parameters

struct pci_dev * dev the PCI device

u32 cap_mask mask of desired AtomicOp sizes, including one or more of: PCI_EXP_DEVCAP2_ATOMIC_COMP32 PCI_EXP_DEVCAP2_ATOMIC_COMP64 PCI_EXP_DEVCAP2_ATOMIC_COMP128

Description

Return 0 if all upstream bridges support AtomicOp routing, egress blocking is disabled on all upstream ports, and the root port supports the requested completion capabilities (32-bit, 64-bit and/or 128-bit AtomicOp completion), or negative otherwise.

u8 **pci_common_swizzle**(struct pci_dev * *dev*, u8 * *pinp*)
swizzle INTx all the way to root bridge

Parameters

struct pci_dev * dev the PCI device

u8 * pinp pointer to the INTx pin value (1=INTA, 2=INTB, 3=INTD, 4=INTD)

Description

Perform INTx swizzling for a device. This traverses through all PCI-to-PCI bridges all the way up to a PCI root bus.

void **`pci_release_region`**(struct pci_dev * *pdev*, int *bar*)
Release a PCI bar

Parameters

`struct pci_dev * pdev` PCI device whose resources were previously reserved by pci_request_region

`int bar` BAR to release

Description

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

int **`pci_request_region`**(struct pci_dev * *pdev*, int *bar*, const char * *res_name*)
Reserve PCI I/O and memory resource

Parameters

`struct pci_dev * pdev` PCI device whose resources are to be reserved

`int bar` BAR to be reserved

`const char * res_name` Name to be associated with resource

Description

Mark the PCI region associated with PCI device **pdev** BAR **bar** as being reserved by owner **res_name**. Do not access any address inside the PCI regions unless this call returns successfully.

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

int **`pci_request_region_exclusive`**(struct pci_dev * *pdev*, int *bar*, const char * *res_name*)
Reserved PCI I/O and memory resource

Parameters

`struct pci_dev * pdev` PCI device whose resources are to be reserved

`int bar` BAR to be reserved

`const char * res_name` Name to be associated with resource.

Description

Mark the PCI region associated with PCI device **pdev** BR **bar** as being reserved by owner **res_name**. Do not access any address inside the PCI regions unless this call returns successfully.

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

The key difference that _exclusive makes it that userspace is explicitly not allowed to map the resource via /dev/mem or sysfs.

void **`pci_release_selected_regions`**(struct pci_dev * *pdev*, int *bars*)
Release selected PCI I/O and memory resources

Parameters

`struct pci_dev * pdev` PCI device whose resources were previously reserved

`int bars` Bitmask of BARs to be released

Description

Release selected PCI I/O and memory resources previously reserved. Call this function only after all use of the PCI regions has ceased.

int **pci_request_selected_regions**(struct pci_dev * *pdev*, int *bars*, const char * *res_name*)
Reserve selected PCI I/O and memory resources

Parameters

struct pci_dev * pdev PCI device whose resources are to be reserved

int bars Bitmask of BARs to be requested

const char * res_name Name to be associated with resource

void **pci_release_regions**(struct pci_dev * *pdev*)
Release reserved PCI I/O and memory resources

Parameters

struct pci_dev * pdev PCI device whose resources were previously reserved by pci_request_regions

Description

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

int **pci_request_regions**(struct pci_dev * *pdev*, const char * *res_name*)
Reserved PCI I/O and memory resources

Parameters

struct pci_dev * pdev PCI device whose resources are to be reserved

const char * res_name Name to be associated with resource.

Description

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

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

int **pci_request_regions_exclusive**(struct pci_dev * *pdev*, const char * *res_name*)
Reserved PCI I/O and memory resources

Parameters

struct pci_dev * pdev PCI device whose resources are to be reserved

const char * res_name Name to be associated with resource.

Description

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

pci_request_regions_exclusive() will mark the region so that /dev/mem and the sysfs MMIO access will not be allowed.

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

int **pci_remap_iospace**(const struct resource * *res*, phys_addr_t *phys_addr*)
Remap the memory mapped I/O space

Parameters

const struct resource * res Resource describing the I/O space

phys_addr_t phys_addr physical address of range to be mapped

Description

Remap the memory mapped I/O space described by the **res** and the CPU physical address **phys_addr** into virtual address space. Only architectures that have memory mapped IO functions defined (and the PCI_IOBASE value defined) should call this function.

void **pci_unmap_iospace**(struct resource * *res*)
Unmap the memory mapped I/O space

Parameters

struct resource * res resource to be unmapped

Description

Unmap the CPU virtual address **res** from virtual address space. Only architectures that have memory mapped IO functions defined (and the PCI_IOBASE value defined) should call this function.

void __iomem * **devm_pci_remap_cfgspace**(struct *device* * *dev*, resource_size_t *offset*, resource_size_t *size*)
Managed pci_remap_cfgspace()

Parameters

struct device * dev Generic device to remap IO address for

resource_size_t offset Resource address to map

resource_size_t size Size of map

Description

Managed pci_remap_cfgspace(). Map is automatically unmapped on driver detach.

void __iomem * **devm_pci_remap_cfg_resource**(struct *device* * *dev*, struct resource * *res*)
check, request region and ioremap cfg resource

Parameters

struct device * dev generic device to handle the resource for

struct resource * res configuration space resource to be handled

Description

Checks that a resource is a valid memory region, requests the memory region and ioremaps with pci_remap_cfgspace() API that ensures the proper PCI configuration space memory attributes are guaranteed.

All operations are managed and will be undone on driver detach.

Returns a pointer to the remapped memory or an ERR_PTR() encoded error code on failure. Usage example:

```
res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
base = devm_pci_remap_cfg_resource(:c:type:`pdev->dev <pdev>`, res);
if (IS_ERR(base))
        return PTR_ERR(base);
```

void **pci_set_master**(struct pci_dev * *dev*)
enables bus-mastering for device dev

Parameters

struct pci_dev * dev the PCI device to enable

Description

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

void **pci_clear_master**(struct pci_dev * *dev*)
disables bus-mastering for device dev

Parameters

struct pci_dev * dev the PCI device to disable

int **pci_set_cacheline_size**(struct pci_dev * *dev*)
ensure the CACHE_LINE_SIZE register is programmed

Parameters

struct pci_dev * dev the PCI device for which MWI is to be enabled

Description

Helper function for pci_set_mwi. Originally copied from drivers/net/acenic.c. Copyright 1998-2001 by Jes Sorensen, <jes**trained**-monkey.org>.

Return

An appropriate -ERRNO error value on error, or zero for success.

int **pci_set_mwi**(struct pci_dev * *dev*)
enables memory-write-invalidate PCI transaction

Parameters

struct pci_dev * dev the PCI device for which MWI is enabled

Description

Enables the Memory-Write-Invalidate transaction in `PCI_COMMAND`.

Return

An appropriate -ERRNO error value on error, or zero for success.

int **pcim_set_mwi**(struct pci_dev * *dev*)
a device-managed *pci_set_mwi()*

Parameters

struct pci_dev * dev the PCI device for which MWI is enabled

Description

Managed *pci_set_mwi()*.

Return

An appropriate -ERRNO error value on error, or zero for success.

int **pci_try_set_mwi**(struct pci_dev * *dev*)
enables memory-write-invalidate PCI transaction

Parameters

struct pci_dev * dev the PCI device for which MWI is enabled

Description

Enables the Memory-Write-Invalidate transaction in `PCI_COMMAND`. Callers are not required to check the return value.

Return

An appropriate -ERRNO error value on error, or zero for success.

void **pci_clear_mwi**(struct pci_dev * *dev*)
disables Memory-Write-Invalidate for device dev

Parameters

struct pci_dev * dev the PCI device to disable

Description

Disables PCI Memory-Write-Invalidate transaction on the device

void **pci_intx**(struct pci_dev * *pdev*, int *enable*)
enables/disables PCI INTx for device dev

Parameters

`struct pci_dev * pdev` the PCI device to operate on

`int enable` boolean: whether to enable or disable PCI INTx

Description

Enables/disables PCI INTx for device dev

bool **`pci_check_and_mask_intx`**(struct pci_dev * *dev*)
mask INTx on pending interrupt

Parameters

`struct pci_dev * dev` the PCI device to operate on

Description

Check if the device dev has its INTx line asserted, mask it and return true in that case. False is returned if no interrupt was pending.

bool **`pci_check_and_unmask_intx`**(struct pci_dev * *dev*)
unmask INTx if no interrupt is pending

Parameters

`struct pci_dev * dev` the PCI device to operate on

Description

Check if the device dev has its INTx line asserted, unmask it if not and return true. False is returned and the mask remains active if there was still an interrupt pending.

int **`pci_wait_for_pending_transaction`**(struct pci_dev * *dev*)
waits for pending transaction

Parameters

`struct pci_dev * dev` the PCI device to operate on

Description

Return 0 if transaction is pending 1 otherwise.

void **`pcie_flr`**(struct pci_dev * *dev*)
initiate a PCIe function level reset

Parameters

`struct pci_dev * dev` device to reset

Description

Initiate a function level reset on **dev**. The caller should ensure the device supports FLR before calling this function, e.g. by using the `pcie_has_flr()` helper.

void **`pci_reset_bridge_secondary_bus`**(struct pci_dev * *dev*)
Reset the secondary bus on a PCI bridge.

Parameters

`struct pci_dev * dev` Bridge device

Description

Use the bridge control register to assert reset on the secondary bus. Devices on the secondary bus are left in power-on state.

int **`__pci_reset_function_locked`**(struct pci_dev * *dev*)
reset a PCI device function while holding the **dev** mutex lock.

Parameters

struct pci_dev * dev PCI device to reset

Description

Some devices allow an individual function to be reset without affecting other functions in the same device. The PCI device must be responsive to PCI config space in order to use this function.

The device function is presumed to be unused and the caller is holding the device mutex lock when this function is called. Resetting the device will make the contents of PCI configuration space random, so any caller of this must be prepared to reinitialise the device including MSI, bus mastering, BARs, decoding IO and memory spaces, etc.

Returns 0 if the device function was successfully reset or negative if the device doesn't support resetting a single function.

int **pci_reset_function**(struct pci_dev * *dev*)
    quiesce and reset a PCI device function

Parameters

struct pci_dev * dev PCI device to reset

Description

Some devices allow an individual function to be reset without affecting other functions in the same device. The PCI device must be responsive to PCI config space in order to use this function.

This function does not just reset the PCI portion of a device, but clears all the state associated with the device. This function differs from *__pci_reset_function_locked()* in that it saves and restores device state over the reset and takes the PCI device lock.

Returns 0 if the device function was successfully reset or negative if the device doesn't support resetting a single function.

int **pci_reset_function_locked**(struct pci_dev * *dev*)
    quiesce and reset a PCI device function

Parameters

struct pci_dev * dev PCI device to reset

Description

Some devices allow an individual function to be reset without affecting other functions in the same device. The PCI device must be responsive to PCI config space in order to use this function.

This function does not just reset the PCI portion of a device, but clears all the state associated with the device. This function differs from *__pci_reset_function_locked()* in that it saves and restores device state over the reset. It also differs from *pci_reset_function()* in that it requires the PCI device lock to be held.

Returns 0 if the device function was successfully reset or negative if the device doesn't support resetting a single function.

int **pci_try_reset_function**(struct pci_dev * *dev*)
    quiesce and reset a PCI device function

Parameters

struct pci_dev * dev PCI device to reset

Description

Same as above, except return -EAGAIN if unable to lock device.

int **pci_probe_reset_slot**(struct pci_slot * *slot*)
    probe whether a PCI slot can be reset

Parameters

struct pci_slot * slot PCI slot to probe

Description

Return 0 if slot can be reset, negative if a slot reset is not supported.

int **pci_reset_slot**(struct pci_slot * *slot*)
reset a PCI slot

Parameters

struct pci_slot * slot PCI slot to reset

Description

A PCI bus may host multiple slots, each slot may support a reset mechanism independent of other slots. For instance, some slots may support slot power control. In the case of a 1:1 bus to slot architecture, this function may wrap the bus reset to avoid spurious slot related events such as hotplug. Generally a slot reset should be attempted before a bus reset. All of the function of the slot and any subordinate buses behind the slot are reset through this function. PCI config space of all devices in the slot and behind the slot is saved before and restored after reset.

Return 0 on success, non-zero on error.

int **pci_try_reset_slot**(struct pci_slot * *slot*)
Try to reset a PCI slot

Parameters

struct pci_slot * slot PCI slot to reset

Description

Same as above except return -EAGAIN if the slot cannot be locked

int **pci_probe_reset_bus**(struct pci_bus * *bus*)
probe whether a PCI bus can be reset

Parameters

struct pci_bus * bus PCI bus to probe

Description

Return 0 if bus can be reset, negative if a bus reset is not supported.

int **pci_reset_bus**(struct pci_bus * *bus*)
reset a PCI bus

Parameters

struct pci_bus * bus top level PCI bus to reset

Description

Do a bus reset on the given bus and any subordinate buses, saving and restoring state of all devices.

Return 0 on success, non-zero on error.

int **pci_try_reset_bus**(struct pci_bus * *bus*)
Try to reset a PCI bus

Parameters

struct pci_bus * bus top level PCI bus to reset

Description

Same as above except return -EAGAIN if the bus cannot be locked

int **pcix_get_max_mmrbc**(struct pci_dev * *dev*)
get PCI-X maximum designed memory read byte count

Parameters

struct pci_dev * dev PCI device to query

Description

Returns mmrbc: maximum designed memory read count in bytes or appropriate error value.

int **pcix_get_mmrbc**(struct pci_dev * *dev*)
    get PCI-X maximum memory read byte count

Parameters

struct pci_dev * dev PCI device to query

Description

Returns mmrbc: maximum memory read count in bytes or appropriate error value.

int **pcix_set_mmrbc**(struct pci_dev * *dev*, int *mmrbc*)
    set PCI-X maximum memory read byte count

Parameters

struct pci_dev * dev PCI device to query

int mmrbc maximum memory read count in bytes valid values are 512, 1024, 2048, 4096

Description

If possible sets maximum memory read byte count, some bridges have erratas that prevent this.

int **pcie_get_readrq**(struct pci_dev * *dev*)
    get PCI Express read request size

Parameters

struct pci_dev * dev PCI device to query

Description

Returns maximum memory read request in bytes or appropriate error value.

int **pcie_set_readrq**(struct pci_dev * *dev*, int *rq*)
    set PCI Express maximum memory read request

Parameters

struct pci_dev * dev PCI device to query

int rq maximum memory read count in bytes valid values are 128, 256, 512, 1024, 2048, 4096

Description

If possible sets maximum memory read request in bytes

int **pcie_get_mps**(struct pci_dev * *dev*)
    get PCI Express maximum payload size

Parameters

struct pci_dev * dev PCI device to query

Description

Returns maximum payload size in bytes

int **pcie_set_mps**(struct pci_dev * *dev*, int *mps*)
    set PCI Express maximum payload size

Parameters

struct pci_dev * dev PCI device to query

int mps maximum payload size in bytes valid values are 128, 256, 512, 1024, 2048, 4096

Description

If possible sets maximum payload size

int **pcie_get_minimum_link**(struct pci_dev * *dev*, enum pci_bus_speed * *speed*, enum pcie_link_width * *width*)
determine minimum link settings of a PCI device

Parameters

struct pci_dev * dev PCI device to query

enum pci_bus_speed * speed storage for minimum speed

enum pcie_link_width * width storage for minimum width

Description

This function will walk up the PCI device chain and determine the minimum link width and speed of the device.

int **pci_select_bars**(struct pci_dev * *dev*, unsigned long *flags*)
Make BAR mask from the type of resource

Parameters

struct pci_dev * dev the PCI device for which BAR mask is made

unsigned long flags resource type mask to be selected

Description

This helper routine makes bar mask from the type of resource.

int **pci_add_dynid**(struct pci_driver * *drv*, unsigned int *vendor*, unsigned int *device*, unsigned int *subvendor*, unsigned int *subdevice*, unsigned int *class*, unsigned int *class_mask*, unsigned long *driver_data*)
add a new PCI device ID to this driver and re-probe devices

Parameters

struct pci_driver * drv target pci driver

unsigned int vendor PCI vendor ID

unsigned int device PCI device ID

unsigned int subvendor PCI subvendor ID

unsigned int subdevice PCI subdevice ID

unsigned int class PCI class

unsigned int class_mask PCI class mask

unsigned long driver_data private driver data

Description

Adds a new dynamic pci device ID to this driver and causes the driver to probe for all devices again. **drv** must have been registered prior to calling this function.

Context

Does GFP_KERNEL allocation.

Return

0 on success, -errno on failure.

const struct pci_device_id * **pci_match_id**(const struct pci_device_id * *ids*, struct pci_dev * *dev*)
See if a pci device matches a given pci_id table

Parameters

const struct pci_device_id * ids array of PCI device id structures to search in

struct pci_dev * dev the PCI device structure to match against.

Description

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

Deprecated, don't use this as it will not catch any dynamic ids that a driver might want to check for.

int **__pci_register_driver**(struct pci_driver * *drv*, struct module * *owner*, const char * *mod_name*)
register a new pci driver

Parameters

struct pci_driver * drv the driver structure to register

struct module * owner owner module of drv

const char * mod_name module name string

Description

Adds the driver structure to the list of registered drivers. Returns a negative value on error, otherwise 0. If no error occurred, the driver remains registered even if no device was claimed during registration.

void **pci_unregister_driver**(struct pci_driver * *drv*)
unregister a pci driver

Parameters

struct pci_driver * drv the driver structure to unregister

Description

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

struct pci_driver * **pci_dev_driver**(const struct pci_dev * *dev*)
get the pci_driver of a device

Parameters

const struct pci_dev * dev the device to query

Description

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

struct pci_dev * **pci_dev_get**(struct pci_dev * *dev*)
increments the reference count of the pci device structure

Parameters

struct pci_dev * dev the device being referenced

Description

Each live reference to a device should be refcounted.

Drivers for PCI devices should normally record such references in their `probe()` methods, when they bind to a device, and release them by calling *`pci_dev_put()`*, in their `disconnect()` methods.

A pointer to the device with the incremented reference counter is returned.

void **pci_dev_put**(struct pci_dev * *dev*)
release a use of the pci device structure

Parameters

struct pci_dev * dev device that's been disconnected

Description

Must be called when a user of a device is finished with it. When the last user of the device calls this function, the memory of the device is freed.

void **`pci_stop_and_remove_bus_device`**(struct pci_dev * *dev*)
remove a PCI device and any children

Parameters

`struct pci_dev * dev` the device to remove

Description

Remove a PCI device from the device lists, informing the drivers that the device has been removed. We also remove any subordinate buses and children in a depth-first manner.

For each device we remove, delete the device structure from the device lists, remove the /proc entry, and notify userspace (/sbin/hotplug).

struct pci_bus * **`pci_find_bus`**(int *domain*, int *busnr*)
locate PCI bus from a given domain and bus number

Parameters

`int domain` number of PCI domain to search

`int busnr` number of desired PCI bus

Description

Given a PCI bus number and domain number, the desired PCI bus is located in the global list of PCI buses. If the bus is found, a pointer to its data structure is returned. If no bus is found, `NULL` is returned.

struct pci_bus * **`pci_find_next_bus`**(const struct pci_bus * *from*)
begin or continue searching for a PCI bus

Parameters

`const struct pci_bus * from` Previous PCI bus found, or `NULL` for new search.

Description

Iterates through the list of known PCI buses. A new search is initiated by passing `NULL` as the **from** argument. Otherwise if **from** is not `NULL`, searches continue from next device on the global list.

struct pci_dev * **`pci_get_slot`**(struct pci_bus * *bus*, unsigned int *devfn*)
locate PCI device for a given PCI slot

Parameters

`struct pci_bus * bus` PCI bus on which desired PCI device resides

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

Description

Given a PCI bus and slot/function number, the desired PCI device is located in the list of PCI devices. If the device is found, its reference count is increased and this function returns a pointer to its data structure. The caller must decrement the reference count by calling *`pci_dev_put()`*. If no device is found, `NULL` is returned.

struct pci_dev * **`pci_get_domain_bus_and_slot`**(int *domain*, unsigned int *bus*, unsigned int *devfn*)
locate PCI device for a given PCI domain (segment), bus, and slot

Parameters

`int domain` PCI domain/segment on which the PCI device resides.

`unsigned int bus` PCI bus on which desired PCI device resides

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

Description

Given a PCI domain, bus, and slot/function number, the desired PCI device is located in the list of PCI devices. If the device is found, its reference count is increased and this function returns a pointer to its data structure. The caller must decrement the reference count by calling `pci_dev_put()`. If no device is found, `NULL` is returned.

struct pci_dev * **pci_get_subsys**(unsigned int *vendor*, unsigned int *device*, unsigned int *ss_vendor*, unsigned int *ss_device*, struct pci_dev * *from*)
begin or continue searching for a PCI device by vendor/subvendor/device/subdevice id

Parameters

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

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

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

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

struct pci_dev * from Previous PCI device found in search, or `NULL` for new search.

Description

Iterates through the list of known PCI devices. If a PCI device is found with a matching **vendor**, **device**, **ss_vendor** and **ss_device**, a pointer to its device structure is returned, and the reference count to the device is incremented. Otherwise, `NULL` is returned. A new search is initiated by passing `NULL` as the **from** argument. Otherwise if **from** is not `NULL`, searches continue from next device on the global list. The reference count for **from** is always decremented if it is not `NULL`.

struct pci_dev * **pci_get_device**(unsigned int *vendor*, unsigned int *device*, struct pci_dev * *from*)
begin or continue searching for a PCI device by vendor/device id

Parameters

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

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

struct pci_dev * from Previous PCI device found in search, or `NULL` for new search.

Description

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

struct pci_dev * **pci_get_class**(unsigned int *class*, struct pci_dev * *from*)
begin or continue searching for a PCI device by class

Parameters

unsigned int class search for a PCI device with this class designation

struct pci_dev * from Previous PCI device found in search, or `NULL` for new search.

Description

Iterates through the list of known PCI devices. If a PCI device is found with a matching **class**, the reference count to the device is incremented and a pointer to its device structure is returned. Otherwise, `NULL` is returned. A new search is initiated by passing `NULL` as the **from** argument. Otherwise if **from** is not `NULL`, searches continue from next device on the global list. The reference count for **from** is always decremented if it is not `NULL`.

int **pci_dev_present**(const struct pci_device_id * *ids*)
Returns 1 if device matching the device list is present, 0 if not.

Parameters

`const struct pci_device_id * ids` A pointer to a null terminated list of struct pci_device_id structures that describe the type of PCI device the caller is trying to find.

Description

Obvious fact: You do not have a reference to any device that might be found by this function, so if that device is removed from the system right after this function is finished, the value will be stale. Use this function to find devices that are usually built into a system, or for a general hint as to if another device happens to be present at this specific moment in time.

void **`pci_msi_mask_irq`**(struct irq_data * *data*)
Generic irq chip callback to mask PCI/MSI interrupts

Parameters

`struct irq_data * data` pointer to irqdata associated to that interrupt

void **`pci_msi_unmask_irq`**(struct irq_data * *data*)
Generic irq chip callback to unmask PCI/MSI interrupts

Parameters

`struct irq_data * data` pointer to irqdata associated to that interrupt

int **`pci_msi_vec_count`**(struct pci_dev * *dev*)
Return the number of MSI vectors a device can send

Parameters

`struct pci_dev * dev` device to report about

Description

This function returns the number of MSI vectors a device requested via Multiple Message Capable register. It returns a negative errno if the device is not capable sending MSI interrupts. Otherwise, the call succeeds and returns a power of two, up to a maximum of 2^5 (32), according to the MSI specification.

int **`pci_msix_vec_count`**(struct pci_dev * *dev*)
return the number of device's MSI-X table entries

Parameters

`struct pci_dev * dev` pointer to the pci_dev data structure of MSI-X device function This function returns the number of device's MSI-X table entries and therefore the number of MSI-X vectors device is capable of sending. It returns a negative errno if the device is not capable of sending MSI-X interrupts.

int **`pci_msi_enabled`**(void)
is MSI enabled?

Parameters

`void` no arguments

Description

Returns true if MSI has not been disabled by the command-line option pci=nomsi.

int **`pci_enable_msix_range`**(struct pci_dev * *dev*, struct msix_entry * *entries*, int *minvec*, int *maxvec*)
configure device's MSI-X capability structure

Parameters

`struct pci_dev * dev` pointer to the pci_dev data structure of MSI-X device function

`struct msix_entry * entries` pointer to an array of MSI-X entries

`int minvec` minimum number of MSI-X irqs requested

`int maxvec` maximum number of MSI-X irqs requested

Description

Setup the MSI-X capability structure of device function with a maximum possible number of interrupts in the range between **minvec** and **maxvec** upon its software driver call to request for MSI-X mode enabled on its hardware device function. It returns a negative errno if an error occurs. If it succeeds, it returns the actual number of interrupts allocated and indicates the successful configuration of MSI-X capability structure with new allocated MSI-X interrupts.

int **pci_alloc_irq_vectors_affinity**(struct pci_dev * *dev*, unsigned int *min_vecs*, unsigned int *max_vecs*, unsigned int *flags*, const struct irq_affinity * *affd*)
allocate multiple IRQs for a device

Parameters

struct pci_dev * dev PCI device to operate on

unsigned int min_vecs minimum number of vectors required (must be >= 1)

unsigned int max_vecs maximum (desired) number of vectors

unsigned int flags flags or quirks for the allocation

const struct irq_affinity * affd optional description of the affinity requirements

Description

Allocate up to **max_vecs** interrupt vectors for **dev**, using MSI-X or MSI vectors if available, and fall back to a single legacy vector if neither is available. Return the number of vectors allocated, (which might be smaller than **max_vecs**) if successful, or a negative error code on error. If less than **min_vecs** interrupt vectors are available for **dev** the function will fail with -ENOSPC.

To get the Linux IRQ number used for a vector that can be passed to `request_irq()` use the *`pci_irq_vector()`* helper.

void **pci_free_irq_vectors**(struct pci_dev * *dev*)
free previously allocated IRQs for a device

Parameters

struct pci_dev * dev PCI device to operate on

Description

Undoes the allocations and enabling in `pci_alloc_irq_vectors()`.

int **pci_irq_vector**(struct pci_dev * *dev*, unsigned int *nr*)
return Linux IRQ number of a device vector

Parameters

struct pci_dev * dev PCI device to operate on

unsigned int nr device-relative interrupt vector index (0-based).

const struct cpumask * **pci_irq_get_affinity**(struct pci_dev * *dev*, int *nr*)
return the affinity of a particular msi vector

Parameters

struct pci_dev * dev PCI device to operate on

int nr device-relative interrupt vector index (0-based).

int **pci_irq_get_node**(struct pci_dev * *pdev*, int *vec*)
return the numa node of a particular msi vector

Parameters

struct pci_dev * pdev PCI device to operate on

int vec device-relative interrupt vector index (0-based).

struct irq_domain * **pci_msi_create_irq_domain**(struct fwnode_handle * *fwnode*, struct msi_domain_info * *info*, struct irq_domain * *parent*)
Create a MSI interrupt domain

Parameters

struct fwnode_handle * fwnode Optional fwnode of the interrupt controller

struct msi_domain_info * info MSI domain info

struct irq_domain * parent Parent irq domain

Description

Updates the domain and chip ops and creates a MSI interrupt domain.

Return

A domain pointer or NULL in case of failure.

int **pci_bus_alloc_resource**(struct pci_bus * *bus*, struct resource * *res*, resource_size_t *size*, resource_size_t *align*, resource_size_t *min*, unsigned long *type_mask*, resource_size_t (*alignf) (void *, const struct resource *, resource_size_t, resource_size_t, void * *alignf_data*)
allocate a resource from a parent bus

Parameters

struct pci_bus * bus PCI bus

struct resource * res resource to allocate

resource_size_t size size of resource to allocate

resource_size_t align alignment of resource to allocate

resource_size_t min minimum /proc/iomem address to allocate

unsigned long type_mask IORESOURCE_* type flags

resource_size_t (*)(void *, const struct resource *, resource_size_t, resource_size_t) alignf
resource alignment function

void * alignf_data data argument for resource alignment function

Description

Given the PCI bus a device resides on, the size, minimum address, alignment and type, try to find an acceptable resource allocation for a specific device resource.

void **pci_bus_add_device**(struct pci_dev * *dev*)
start driver for a single device

Parameters

struct pci_dev * dev device to add

Description

This adds add sysfs entries and start device drivers

void **pci_bus_add_devices**(const struct pci_bus * *bus*)
start driver for PCI devices

Parameters

const struct pci_bus * bus bus to check for new devices

Description

Start driver for PCI devices and add some sysfs entries.

struct pci_ops * **pci_bus_set_ops**(struct pci_bus * *bus*, struct pci_ops * *ops*)
Set raw operations of pci bus

Parameters

struct pci_bus * bus pci bus struct

struct pci_ops * ops new raw operations

Description

Return previous raw operations

ssize_t **pci_read_vpd**(struct pci_dev * *dev*, loff_t *pos*, size_t *count*, void * *buf*)
Read one entry from Vital Product Data

Parameters

struct pci_dev * dev pci device struct

loff_t pos offset in vpd space

size_t count number of bytes to read

void * buf pointer to where to store result

ssize_t **pci_write_vpd**(struct pci_dev * *dev*, loff_t *pos*, size_t *count*, const void * *buf*)
Write entry to Vital Product Data

Parameters

struct pci_dev * dev pci device struct

loff_t pos offset in vpd space

size_t count number of bytes to write

const void * buf buffer containing write data

int **pci_set_vpd_size**(struct pci_dev * *dev*, size_t *len*)
Set size of Vital Product Data space

Parameters

struct pci_dev * dev pci device struct

size_t len size of vpd space

void **pci_cfg_access_lock**(struct pci_dev * *dev*)
Lock PCI config reads/writes

Parameters

struct pci_dev * dev pci device struct

Description

When access is locked, any userspace reads or writes to config space and concurrent lock requests will sleep until access is allowed via *pci_cfg_access_unlock()* again.

bool **pci_cfg_access_trylock**(struct pci_dev * *dev*)
try to lock PCI config reads/writes

Parameters

struct pci_dev * dev pci device struct

Description

Same as pci_cfg_access_lock, but will return 0 if access is already locked, 1 otherwise. This function can be used from atomic contexts.

void **pci_cfg_access_unlock**(struct pci_dev * *dev*)
Unlock PCI config reads/writes

Parameters

`struct pci_dev * dev` pci device struct

Description

This function allows PCI config accesses to resume.

enum pci_lost_interrupt_reason **`pci_lost_interrupt`**(struct pci_dev * *pdev*)
reports a lost PCI interrupt

Parameters

`struct pci_dev * pdev` device whose interrupt is lost

Description

The primary function of this routine is to report a lost interrupt in a standard way which users can recognise (instead of blaming the driver).

Return

a suggestion for fixing it (although the driver is not required to act on this).

int **`pci_request_irq`**(struct pci_dev * *dev*, unsigned int *nr*, irq_handler_t *handler*, irq_handler_t *thread_fn*, void * *dev_id*, const char * *fmt*, ...)
allocate an interrupt line for a PCI device

Parameters

`struct pci_dev * dev` PCI device to operate on

`unsigned int nr` device-relative interrupt vector index (0-based).

`irq_handler_t handler` Function to be called when the IRQ occurs. Primary handler for threaded interrupts. If NULL and thread_fn != NULL the default primary handler is installed.

`irq_handler_t thread_fn` Function called from the IRQ handler thread If NULL, no IRQ thread is created

`void * dev_id` Cookie passed back to the handler function

`const char * fmt` Printf-like format string naming the handler

`...` variable arguments

Description

This call allocates interrupt resources and enables the interrupt line and IRQ handling. From the point this call is made **handler** and **thread_fn** may be invoked. All interrupts requested using this function might be shared.

dev_id must not be NULL and must be globally unique.

void **`pci_free_irq`**(struct pci_dev * *dev*, unsigned int *nr*, void * *dev_id*)
free an interrupt allocated with pci_request_irq

Parameters

`struct pci_dev * dev` PCI device to operate on

`unsigned int nr` device-relative interrupt vector index (0-based).

`void * dev_id` Device identity to free

Description

Remove an interrupt handler. The handler is removed and if the interrupt line is no longer in use by any driver it is disabled. The caller must ensure the interrupt is disabled on the device before calling this function. The function does not return until any executing interrupts for this IRQ have completed.

This function must not be called from interrupt context.

bool **`pcie_relaxed_ordering_enabled`**(struct pci_dev * *dev*)
Probe for PCIe relaxed ordering enable

Parameters

`struct pci_dev * dev` PCI device to query

Description

Returns true if the device has enabled relaxed ordering attribute.

int **`pci_scan_slot`**(struct pci_bus * *bus*, int *devfn*)
Scan a PCI slot on a bus for devices

Parameters

`struct pci_bus * bus` PCI bus to scan

`int devfn` slot number to scan (must have zero function)

Description

Scan a PCI slot on the specified PCI bus for devices, adding discovered devices to the **bus**->devices list. New devices will not have is_added set.

Returns the number of new devices found.

unsigned int **`pci_scan_child_bus`**(struct pci_bus * *bus*)
Scan devices below a bus

Parameters

`struct pci_bus * bus` Bus to scan for devices

Description

Scans devices below **bus** including subordinate buses. Returns new subordinate number including all the found devices.

unsigned int **`pci_rescan_bus`**(struct pci_bus * *bus*)
Scan a PCI bus for devices

Parameters

`struct pci_bus * bus` PCI bus to scan

Description

Scan a PCI bus and child buses for new devices, add them, and enable them.

Returns the max number of subordinate bus discovered.

struct pci_slot * **`pci_create_slot`**(struct pci_bus * *parent*, int *slot_nr*, const char * *name*, struct hotplug_slot * *hotplug*)
create or increment refcount for physical PCI slot

Parameters

`struct pci_bus * parent` struct pci_bus of parent bridge

`int slot_nr` PCI_SLOT(pci_dev->devfn) or -1 for placeholder

`const char * name` user visible string presented in /sys/bus/pci/slots/<name>

`struct hotplug_slot * hotplug` set if caller is hotplug driver, NULL otherwise

Description

PCI slots have first class attributes such as address, speed, width, and a `struct pci_slot` is used to manage them. This interface will either return a new `struct pci_slot` to the caller, or if the pci_slot already exists, its refcount will be incremented.

Slots are uniquely identified by a **pci_bus**, **slot_nr** tuple.

There are known platforms with broken firmware that assign the same name to multiple slots. Workaround these broken platforms by renaming the slots on behalf of the caller. If firmware assigns name N to multiple slots:

The first slot is assigned N The second slot is assigned N-1 The third slot is assigned N-2 etc.

Placeholder slots: In most cases, **pci_bus**, **slot_nr** will be sufficient to uniquely identify a slot. There is one notable exception - pSeries (rpaphp), where the **slot_nr** cannot be determined until a device is actually inserted into the slot. In this scenario, the caller may pass -1 for **slot_nr**.

The following semantics are imposed when the caller passes **slot_nr** == -1. First, we no longer check for an existing `struct` pci_slot, as there may be many slots with **slot_nr** of -1. The other change in semantics is user-visible, which is the 'address' parameter presented in sysfs will consist solely of a dddd:bb tuple, where dddd is the PCI domain of the `struct` pci_bus and bb is the bus number. In other words, the devfn of the 'placeholder' slot will not be displayed.

void **pci_destroy_slot**(struct pci_slot * *slot*)
decrement refcount for physical PCI slot

Parameters

struct pci_slot * slot struct pci_slot to decrement

Description

`struct` pci_slot is refcounted, so destroying them is really easy; we just call kobject_put on its kobj and let our release methods do the rest.

void **pci_hp_create_module_link**(struct pci_slot * *pci_slot*)
create symbolic link to the hotplug driver module.

Parameters

struct pci_slot * pci_slot struct pci_slot

Description

Helper function for pci_hotplug_core.c to create symbolic link to the hotplug driver module.

void **pci_hp_remove_module_link**(struct pci_slot * *pci_slot*)
remove symbolic link to the hotplug driver module.

Parameters

struct pci_slot * pci_slot struct pci_slot

Description

Helper function for pci_hotplug_core.c to remove symbolic link to the hotplug driver module.

int **pci_enable_rom**(struct pci_dev * *pdev*)
enable ROM decoding for a PCI device

Parameters

struct pci_dev * pdev PCI device to enable

Description

Enable ROM decoding on **dev**. This involves simply turning on the last bit of the PCI ROM BAR. Note that some cards may share address decoders between the ROM and other resources, so enabling it may disable access to MMIO registers or other card memory.

void **pci_disable_rom**(struct pci_dev * *pdev*)
disable ROM decoding for a PCI device

Parameters

struct pci_dev * pdev PCI device to disable

Description

Disable ROM decoding on a PCI device by turning off the last bit in the ROM BAR.

void __iomem * **pci_map_rom**(struct pci_dev * *pdev*, size_t * *size*)
map a PCI ROM to kernel space

Parameters

struct pci_dev * pdev pointer to pci device struct

size_t * size pointer to receive size of pci window over ROM

Return

kernel virtual pointer to image of ROM

Map a PCI ROM into kernel space. If ROM is boot video ROM, the shadow BIOS copy will be returned instead of the actual ROM.

void **pci_unmap_rom**(struct pci_dev * *pdev*, void __iomem * *rom*)
unmap the ROM from kernel space

Parameters

struct pci_dev * pdev pointer to pci device struct

void __iomem * rom virtual address of the previous mapping

Description

Remove a mapping of a previously mapped ROM

void __iomem * **pci_platform_rom**(struct pci_dev * *pdev*, size_t * *size*)
provides a pointer to any ROM image provided by the platform

Parameters

struct pci_dev * pdev pointer to pci device struct

size_t * size pointer to receive size of pci window over ROM

int **pci_enable_sriov**(struct pci_dev * *dev*, int *nr_virtfn*)
enable the SR-IOV capability

Parameters

struct pci_dev * dev the PCI device

int nr_virtfn number of virtual functions to enable

Description

Returns 0 on success, or negative on failure.

void **pci_disable_sriov**(struct pci_dev * *dev*)
disable the SR-IOV capability

Parameters

struct pci_dev * dev the PCI device

int **pci_num_vf**(struct pci_dev * *dev*)
return number of VFs associated with a PF device_release_driver

Parameters

struct pci_dev * dev the PCI device

Description

Returns number of VFs, or 0 if SR-IOV is not enabled.

int **pci_vfs_assigned**(struct pci_dev * *dev*)
returns number of VFs are assigned to a guest

Parameters

struct pci_dev * dev the PCI device

Description

Returns number of VFs belonging to this device that are assigned to a guest. If device is not a physical function returns 0.

int **pci_sriov_set_totalvfs**(struct pci_dev * *dev*, u16 *numvfs*)

- reduce the TotalVFs available

Parameters

struct pci_dev * dev the PCI PF device

u16 numvfs number that should be used for TotalVFs supported

Description

Should be called from PF driver's probe routine with device's mutex held.

Returns 0 if PF is an SRIOV-capable device and value of numvfs valid. If not a PF return -ENOSYS; if numvfs is invalid return -EINVAL; if VFs already enabled, return -EBUSY.

int **pci_sriov_get_totalvfs**(struct pci_dev * *dev*)

- get total VFs supported on this device

Parameters

struct pci_dev * dev the PCI PF device

Description

For a PCIe device with SRIOV support, return the PCIe SRIOV capability value of TotalVFs or the value of driver_max_VFs if the driver reduced it. Otherwise 0.

ssize_t **pci_read_legacy_io**(struct file * *filp*, struct kobject * *kobj*, struct bin_attribute * *bin_attr*, char * *buf*, loff_t *off*, size_t *count*)
read byte(s) from legacy I/O port space

Parameters

struct file * filp open sysfs file

struct kobject * kobj kobject corresponding to file to read from

struct bin_attribute * bin_attr struct bin_attribute for this file

char * buf buffer to store results

loff_t off offset into legacy I/O port space

size_t count number of bytes to read

Description

Reads 1, 2, or 4 bytes from legacy I/O port space using an arch specific callback routine (pci_legacy_read).

ssize_t **pci_write_legacy_io**(struct file * *filp*, struct kobject * *kobj*, struct bin_attribute * *bin_attr*, char * *buf*, loff_t *off*, size_t *count*)
write byte(s) to legacy I/O port space

Parameters

struct file * filp open sysfs file

struct kobject * kobj kobject corresponding to file to read from

struct bin_attribute * bin_attr struct bin_attribute for this file

char * buf buffer containing value to be written

loff_t off offset into legacy I/O port space

size_t count number of bytes to write

Description

Writes 1, 2, or 4 bytes from legacy I/O port space using an arch specific callback routine (pci_legacy_write).

int **pci_mmap_legacy_mem**(struct file * *filp*, struct kobject * *kobj*, struct bin_attribute * *attr*, struct vm_area_struct * *vma*)
map legacy PCI memory into user memory space

Parameters

struct file * filp open sysfs file

struct kobject * kobj kobject corresponding to device to be mapped

struct bin_attribute * attr struct bin_attribute for this file

struct vm_area_struct * vma struct vm_area_struct passed to mmap

Description

Uses an arch specific callback, pci_mmap_legacy_mem_page_range, to mmap legacy memory space (first meg of bus space) into application virtual memory space.

int **pci_mmap_legacy_io**(struct file * *filp*, struct kobject * *kobj*, struct bin_attribute * *attr*, struct vm_area_struct * *vma*)
map legacy PCI IO into user memory space

Parameters

struct file * filp open sysfs file

struct kobject * kobj kobject corresponding to device to be mapped

struct bin_attribute * attr struct bin_attribute for this file

struct vm_area_struct * vma struct vm_area_struct passed to mmap

Description

Uses an arch specific callback, pci_mmap_legacy_io_page_range, to mmap legacy IO space (first meg of bus space) into application virtual memory space. Returns -ENOSYS if the operation isn't supported

void **pci_adjust_legacy_attr**(struct pci_bus * *b*, enum pci_mmap_state *mmap_type*)
adjustment of legacy file attributes

Parameters

struct pci_bus * b bus to create files under

enum pci_mmap_state mmap_type I/O port or memory

Description

Stub implementation. Can be overridden by arch if necessary.

void **pci_create_legacy_files**(struct pci_bus * *b*)
create legacy I/O port and memory files

Parameters

struct pci_bus * b bus to create files under

Description

Some platforms allow access to legacy I/O port and ISA memory space on a per-bus basis. This routine creates the files and ties them into their associated read, write and mmap files from pci-sysfs.c

On error unwind, but don't propagate the error to the caller as it is ok to set up the PCI bus without these files.

int **pci_mmap_resource**(struct kobject * *kobj*, struct bin_attribute * *attr*, struct vm_area_struct * *vma*, int *write_combine*)
map a PCI resource into user memory space

Parameters

struct kobject * kobj kobject for mapping

struct bin_attribute * attr struct bin_attribute for the file being mapped

struct vm_area_struct * vma struct vm_area_struct passed into the mmap

int write_combine 1 for write_combine mapping

Description

Use the regular PCI mapping routines to map a PCI resource into userspace.

void **pci_remove_resource_files**(struct pci_dev * *pdev*)
cleanup resource files

Parameters

struct pci_dev * pdev dev to cleanup

Description

If we created resource files for **pdev**, remove them from sysfs and free their resources.

int **pci_create_resource_files**(struct pci_dev * *pdev*)
create resource files in sysfs for **dev**

Parameters

struct pci_dev * pdev dev in question

Description

Walk the resources in **pdev** creating files for each resource available.

ssize_t **pci_write_rom**(struct file * *filp*, struct kobject * *kobj*, struct bin_attribute * *bin_attr*, char * *buf*, loff_t *off*, size_t *count*)
used to enable access to the PCI ROM display

Parameters

struct file * filp sysfs file

struct kobject * kobj kernel object handle

struct bin_attribute * bin_attr struct bin_attribute for this file

char * buf user input

loff_t off file offset

size_t count number of byte in input

Description

writing anything except 0 enables it

ssize_t **pci_read_rom**(struct file * *filp*, struct kobject * *kobj*, struct bin_attribute * *bin_attr*, char * *buf*, loff_t *off*, size_t *count*)
read a PCI ROM

Parameters

struct file * filp sysfs file

struct kobject * kobj kernel object handle

struct bin_attribute * bin_attr struct bin_attribute for this file

char * buf where to put the data we read from the ROM

`loff_t off` file offset

`size_t count` number of bytes to read

Description

Put **count** bytes starting at **off** into **buf** from the ROM in the PCI device corresponding to **kobj**.

void **`pci_remove_sysfs_dev_files`**(struct pci_dev * *pdev*)
cleanup PCI specific sysfs files

Parameters

`struct pci_dev * pdev` device whose entries we should free

Description

Cleanup when **pdev** is removed from sysfs.

CHAPTER

FIFTEEN

PCI HOTPLUG SUPPORT LIBRARY

int **__pci_hp_register**(struct hotplug_slot * *slot*, struct pci_bus * *bus*, int *devnr*, const char * *name*, struct module * *owner*, const char * *mod_name*)
register a hotplug_slot with the PCI hotplug subsystem

Parameters

struct hotplug_slot * slot pointer to the `struct hotplug_slot` to register

struct pci_bus * bus bus this slot is on

int devnr device number

const char * name name registered with kobject core

struct module * owner caller module owner

const char * mod_name caller module name

Description

Registers a hotplug slot with the pci hotplug subsystem, which will allow userspace interaction to the slot.

Returns 0 if successful, anything else for an error.

int **pci_hp_deregister**(struct hotplug_slot * *slot*)
deregister a hotplug_slot with the PCI hotplug subsystem

Parameters

struct hotplug_slot * slot pointer to the `struct hotplug_slot` to deregister

Description

The **slot** must have been registered with the pci hotplug subsystem previously with a call to `pci_hp_register()`.

Returns 0 if successful, anything else for an error.

int **pci_hp_change_slot_info**(struct hotplug_slot * *slot*, struct hotplug_slot_info * *info*)
changes the slot's information structure in the core

Parameters

struct hotplug_slot * slot pointer to the slot whose info has changed

struct hotplug_slot_info * info pointer to the info copy into the slot's info structure

Description

slot must have been registered with the pci hotplug subsystem previously with a call to `pci_hp_register()`.

Returns 0 if successful, anything else for an error.

CHAPTER

SIXTEEN

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.

struct **spi_statistics**
statistics for spi transfers

Definition

```
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];
  unsigned long transfers_split_maxsize;
};
```

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 transfer bytes histogramm

transfers_split_maxsize number of transfers that have been split because of maxsize limit

struct **spi_device**
Controller side proxy for an SPI slave device

Definition

```
struct spi_device {
  struct device           dev;
  struct spi_controller   *controller;
  struct spi_controller   *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.

controller SPI controller used with the device.

master Copy of controller, for backwards compatibility.

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 **controller**.

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 spi_transfer.bits_per_word can override this for each transfer.

mode The spi 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 SPI_CS_HIGH) as can the "MSB first" default for each word in a transfer (by specifying SPI_LSB_FIRST).

irq Negative, or the number passed to `request_irq()` to receive interrupts from this device.

controller_state Controller's runtime state

controller_data Board-specific definitions for controller, such as FIFO initialization parameters; from board_info.controller_data

modalias 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

cs_gpio gpio number of the chipselect line (optional, -ENOENT when not using a GPIO line)

statistics statistics for the spi_device

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.

struct **spi_driver**
Host side "protocol" driver

Definition

```
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.

void **spi_unregister_driver**(struct *spi_driver* * *sdrv*)
reverse effect of spi_register_driver

Parameters

struct spi_driver * sdrv the driver to unregister

Context

can sleep

module_spi_driver(*__spi_driver*)
Helper macro for registering a SPI driver

Parameters

__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()*

struct **spi_controller**
interface to SPI master or slave controller

Definition

```
struct spi_controller {
  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_CONTROLLER_HALF_DUPLEX      BIT(0)  ;
#define SPI_CONTROLLER_NO_RX            BIT(1)  ;
#define SPI_CONTROLLER_NO_TX            BIT(2)  ;
#define SPI_CONTROLLER_MUST_RX          BIT(3)  ;
#define SPI_CONTROLLER_MUST_TX          BIT(4)  ;
#define SPI_MASTER_GPIO_SS              BIT(5)  ;
  bool slave;
  size_t (*max_transfer_size)(struct spi_device *spi);
  size_t (*max_message_size)(struct spi_device *spi);
  struct mutex            io_mutex;
  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_controller *ctlr,struct spi_device *spi, struct spi_transfer *xfer);
  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_controller *ctlr);
  int (*transfer_one_message)(struct spi_controller *ctlr, struct spi_message *mesg);
  int (*unprepare_transfer_hardware)(struct spi_controller *ctlr);
  int (*prepare_message)(struct spi_controller *ctlr, struct spi_message *message);
  int (*unprepare_message)(struct spi_controller *ctlr, struct spi_message *message);
  int (*slave_abort)(struct spi_controller *ctlr);
  int (*spi_flash_read)(struct  spi_device *spi, struct spi_flash_read_message *msg);
  bool (*spi_flash_can_dma)(struct spi_device *spi, struct spi_flash_read_message *msg);
  bool (*flash_read_supported)(struct spi_device *spi);
  void (*set_cs)(struct spi_device *spi, bool enable);
  int (*transfer_one)(struct spi_controller *ctlr, struct spi_device *spi, struct spi_transfer *transfer
  void (*handle_err)(struct spi_controller *ctlr, 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;
  int (*fw_translate_cs)(struct spi_controller *ctlr, unsigned cs);
};
```

Members

dev device interface to this driver

list link with the global spi_controller 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

slave indicates that this is an SPI slave controller

max_transfer_size function that returns the max transfer size for a *spi_device*; may be NULL, so the default SIZE_MAX will be used.

max_message_size function that returns the max message size for a *spi_device*; may be NULL, so the default SIZE_MAX will be used.

io_mutex mutex for physical bus access

bus_lock_spinlock spinlock for SPI bus locking

bus_lock_mutex mutex for exclusion of multiple callers

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

can_dma determine whether this controller supports DMA

queued whether this controller is providing an internal message queue

kworker thread struct for message pump

kworker_task pointer to task for message pump kworker thread

pump_messages work struct for scheduling work to the message pump

queue_lock spinlock to syncronise access to message queue

queue message queue

cur_msg the currently in-flight message

idling the device is entering idle state

busy message pump is busy

running message pump is running

rt whether this queue is set to run as a realtime task

auto_runtime_pm the core should ensure a runtime PM reference is held while the hardware is prepared, using the parent device for the spidev

cur_msg_prepared spi_prepare_message was called for the currently in-flight message

cur_msg_mapped message has been mapped for DMA

xfer_completion used by core `transfer_one_message()`

max_dma_len Maximum length of a DMA transfer for the device.

prepare_transfer_hardware a message will soon arrive from the queue so the subsystem requests the driver to prepare the transfer hardware by issuing this call

transfer_one_message 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 *spi_finalize_current_message()* so the subsystem can issue the next message

unprepare_transfer_hardware 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

prepare_message set up the controller to transfer a single message, for example doing DMA mapping. Called from threaded context.

unprepare_message undo any work done by `prepare_message()`.

slave_abort abort the ongoing transfer request on an SPI slave controller

spi_flash_read to support spi-controller hardwares that provide accelerated interface to read from flash devices.

spi_flash_can_dma analogous to `can_dma()` interface, but for controllers implementing spi_flash_read.

flash_read_supported spi device supports flash read

set_cs set the logic level of the chip select line. May be called from interrupt context.

transfer_one transfer a single spi_transfer. - 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 *spi_finalize_current_transfer()* so the subsystem can issue the next transfer. Note: transfer_one and transfer_one_message are mutually exclusive; when both are set, the generic subsystem does not call your transfer_one callback.

handle_err the subsystem calls the driver to handle an error that occurs in the generic implementation of `transfer_one_message()`.

cs_gpios Array of GPIOs to use as chip select lines; one per CS number. Any individual value may be -ENOENT for CS lines that are not GPIOs (driven by the SPI controller itself).

statistics statistics for the spi_controller

dma_tx DMA transmit channel

dma_rx DMA receive channel

dummy_rx dummy receive buffer for full-duplex devices

dummy_tx dummy transmit buffer for full-duplex devices

fw_translate_cs If the boot firmware uses different numbering scheme what Linux expects, this optional hook can be used to translate between the two.

Description

Each SPI 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.

struct **spi_res**
spi resource management structure

Definition

```
struct spi_res {
  struct list_head        entry;
  spi_res_release_t release;
  unsigned long long      data[];
};
```

Members

entry list entry

release release code called prior to freeing this resource

data extra data allocated for the specific use-case

Description

this is based on ideas from devres, but focused on life-cycle management during spi_message processing

struct **spi_transfer**
a read/write buffer pair

Definition

```
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 **spi_message.is_dma_mapped**

rx_dma DMA address of rx_buf, if **spi_message.is_dma_mapped**

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 **spi_device**) is used.

delay_usecs microseconds to delay after this transfer before (optionally) changing the chipselect status, then starting the next transfer or completing this **spi_message**.

speed_hz Select a speed other than the device default for this transfer. If 0 the default (from **spi_device**) is used.

transfer_list transfers are sequenced through **spi_message.transfers**

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 intactive.

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.

struct **`spi_message`**
    one multi-segment SPI transaction

Definition

```
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;
  struct list_head        resources;
};
```

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 `complete()` 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

resources for resource management when the spi message is processed

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.

void **spi_message_init_with_transfers**(struct *spi_message* * *m*, struct *spi_transfer* * *xfers*, unsigned int *num_xfers*)
Initialize spi_message and append transfers

Parameters

struct spi_message * m spi_message to be initialized

struct spi_transfer * xfers An array of spi transfers

unsigned int num_xfers 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.

struct **spi_replaced_transfers**
structure describing the spi_transfer replacements that have occurred so that they can get reverted

Definition

```
struct spi_replaced_transfers {
  spi_replaced_release_t release;
  void *extradata;
  struct list_head replaced_transfers;
  struct list_head *replaced_after;
  size_t inserted;
  struct spi_transfer inserted_transfers[];
};
```

Members

release some extra release code to get executed prior to relasing this structure

extradata pointer to some extra data if requested or NULL

replaced_transfers transfers that have been replaced and which need to get restored

replaced_after the transfer after which the **replaced_transfers** are to get re-inserted

inserted number of transfers inserted

inserted_transfers array of spi_transfers of array-size **inserted**, that have been replacing replaced_transfers

note

that **extradata** will point to **inserted_transfers**[**inserted**] if some extra allocation is requested, so alignment will be the same as for spi_transfers

int **spi_sync_transfer**(struct *spi_device* * *spi*, struct *spi_transfer* * *xfers*, unsigned int *num_xfers*)
synchronous SPI data transfer

Parameters

struct spi_device * spi device with which data will be exchanged

struct spi_transfer * xfers An array of spi_transfers

unsigned int 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.

int **spi_write**(struct *spi_device* * *spi*, const void * *buf*, size_t *len*)
SPI synchronous write

Parameters

struct spi_device * spi device to which data will be written

const void * buf data buffer

size_t 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.

int **spi_read**(struct *spi_device* * *spi*, void * *buf*, size_t *len*)
SPI synchronous read

Parameters

struct spi_device * spi device from which data will be read

void * buf data buffer

size_t 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.

ssize_t **spi_w8r8**(struct *spi_device* * *spi*, u8 *cmd*)
SPI synchronous 8 bit write followed by 8 bit read

Parameters

`struct spi_device * spi` device with which data will be exchanged

`u8 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.

ssize_t **`spi_w8r16`**(struct *spi_device* * *spi*, u8 *cmd*)
SPI synchronous 8 bit write followed by 16 bit read

Parameters

`struct spi_device * spi` device with which data will be exchanged

`u8 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.

ssize_t **`spi_w8r16be`**(struct *spi_device* * *spi*, u8 *cmd*)
SPI synchronous 8 bit write followed by 16 bit big-endian read

Parameters

`struct spi_device * spi` device with which data will be exchanged

`u8 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.

struct **`spi_flash_read_message`**
flash specific information for spi-masters that provide accelerated flash read interfaces

Definition

```
struct spi_flash_read_message {
  void *buf;
  loff_t from;
  size_t len;
  size_t retlen;
  u8 read_opcode;
  u8 addr_width;
  u8 dummy_bytes;
  u8 opcode_nbits;
  u8 addr_nbits;
  u8 data_nbits;
  struct sg_table rx_sg;
  bool cur_msg_mapped;
};
```

Members

`buf` buffer to read data

`from` offset within the flash from where data is to be read

`len` length of data to be read

`retlen` actual length of data read

`read_opcode` read_opcode to be used to communicate with flash

`addr_width` number of address bytes

`dummy_bytes` number of dummy bytes

`opcode_nbits` number of lines to send opcode

`addr_nbits` number of lines to send address

`data_nbits` number of lines for data

`rx_sg` Scatterlist for receive data read from flash

`cur_msg_mapped` message has been mapped for DMA

struct **`spi_board_info`**
board-specific template for a SPI device

Definition

```
struct spi_board_info {
  char modalias[SPI_NAME_SIZE];
  const void      *platform_data;
  const struct property_entry *properties;
  void *controller_data;
  int irq;
  u32 max_speed_hz;
  u16 bus_num;
  u16 chip_select;
  u16 mode;
};
```

Members

`modalias` Initializes spi_device.modalias; identifies the driver.

`platform_data` Initializes spi_device.platform_data; the particular data stored there is driver-specific.

`properties` Additional device properties for the device.

`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_controller 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.

int **spi_register_board_info**(struct *spi_board_info* const * *info*, unsigned *n*)
register SPI devices for a given board

Parameters

struct spi_board_info const * info array of chip descriptors

unsigned 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. Device properties are deep-copied though.

Return

zero on success, else a negative error code.

int **__spi_register_driver**(struct module * *owner*, struct *spi_driver* * *sdrv*)
register a SPI driver

Parameters

struct module * owner owner module of the driver to register

struct spi_driver * sdrv the driver to register

Context

can sleep

Return

zero on success, else a negative error code.

struct *spi_device* * **spi_alloc_device**(struct *spi_controller* * *ctlr*)
Allocate a new SPI device

Parameters

`struct spi_controller * ctlr` 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 controller. 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.

int **`spi_add_device`**(struct *spi_device* * *spi*)
    Add spi_device allocated with spi_alloc_device

Parameters

`struct spi_device * 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

struct *spi_device* * **`spi_new_device`**(struct *spi_controller* * *ctlr*, struct *spi_board_info* * *chip*)
    instantiate one new SPI device

Parameters

`struct spi_controller * ctlr` Controller to which device is connected

`struct spi_board_info * 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.

void **`spi_unregister_device`**(struct *spi_device* * *spi*)
    unregister a single SPI device

Parameters

`struct spi_device * spi` spi_device to unregister

Description

Start making the passed SPI device vanish. Normally this would be handled by *`spi_unregister_controller()`*.

void **spi_finalize_current_transfer**(struct *spi_controller* * *ctlr*)
report completion of a transfer

Parameters

struct spi_controller * ctlr the controller 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.

struct *spi_message* * **spi_get_next_queued_message**(struct *spi_controller* * *ctlr*)
called by driver to check for queued messages

Parameters

struct spi_controller * ctlr the controller 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.

void **spi_finalize_current_message**(struct *spi_controller* * *ctlr*)
the current message is complete

Parameters

struct spi_controller * ctlr the controller 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.

int **spi_slave_abort**(struct *spi_device* * *spi*)
abort the ongoing transfer request on an SPI slave controller

Parameters

struct spi_device * spi device used for the current transfer

struct *spi_controller* * **__spi_alloc_controller**(struct *device* * *dev*, unsigned int *size*, bool *slave*)
allocate an SPI master or slave controller

Parameters

struct device * dev the controller, possibly using the platform_bus

unsigned int 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_controller_get_devdata()`.

bool slave flag indicating whether to allocate an SPI master (false) or SPI slave (true) controller

Context

can sleep

Description

This call is used only by SPI controller drivers, which are the only ones directly touching chip registers. It's how they allocate an spi_controller structure, prior to calling *`spi_register_controller()`*.

This must be called from context that can sleep.

The caller is responsible for assigning the bus number and initializing the controller's methods before calling *`spi_register_controller()`*; and (after errors adding the device) calling `spi_controller_put()` to prevent a memory leak.

Return

the SPI controller structure on success, else NULL.

int **spi_register_controller**(struct *spi_controller* * *ctlr*)
register SPI master or slave controller

Parameters

struct spi_controller * ctlr initialized master, originally from `spi_alloc_master()` or `spi_alloc_slave()`

Context

can sleep

Description

SPI 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_controller()* 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 controller's refcount). After a successful return, the caller is responsible for calling *spi_unregister_controller()*.

Return

zero on success, else a negative error code.

int **devm_spi_register_controller**(struct *device* * *dev*, struct *spi_controller* * *ctlr*)
register managed SPI master or slave controller

Parameters

struct device * dev device managing SPI controller

struct spi_controller * ctlr initialized controller, originally from `spi_alloc_master()` or `spi_alloc_slave()`

Context

can sleep

Description

Register a SPI device as with *spi_register_controller()* which will automatically be unregistered and freed.

Return

zero on success, else a negative error code.

void **spi_unregister_controller**(struct *spi_controller* * *ctlr*)
unregister SPI master or slave controller

Parameters

struct spi_controller * ctlr the controller being unregistered

Context

can sleep

Description

This call is used only by SPI controller drivers, which are the only ones directly touching chip registers.

This must be called from context that can sleep.

Note that this function also drops a reference to the controller.

struct *spi_controller* * **spi_busnum_to_master**(u16 *bus_num*)
look up master associated with bus_num

Parameters

u16 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_controller (which the caller must release), or NULL if there is no such master registered.

Return

the SPI master structure on success, else NULL.

void * **spi_res_alloc**(struct *spi_device* * *spi*, spi_res_release_t *release*, size_t *size*, gfp_t *gfp*)
allocate a spi resource that is life-cycle managed during the processing of a spi_message while using spi_transfer_one

Parameters

struct spi_device * spi the spi device for which we allocate memory

spi_res_release_t release the release code to execute for this resource

size_t size size to alloc and return

gfp_t gfp GFP allocation flags

Return

the pointer to the allocated data

This may get enhanced in the future to allocate from a memory pool of the **spi_device** or **spi_controller** to avoid repeated allocations.

void **spi_res_free**(void * *res*)
free an spi resource

Parameters

void * res pointer to the custom data of a resource

void **spi_res_add**(struct *spi_message* * *message*, void * *res*)
add a spi_res to the spi_message

Parameters

struct spi_message * message the spi message

void * res the spi_resource

void **spi_res_release**(struct *spi_controller* * *ctlr*, struct *spi_message* * *message*)
release all spi resources for this message

Parameters

struct spi_controller * ctlr the **spi_controller**

struct spi_message * message the **spi_message**

struct *spi_replaced_transfers* * **spi_replace_transfers**(struct *spi_message* * *msg*, struct *spi_transfer* * *xfer_first*, size_t *remove*, size_t *insert*, spi_replaced_release_t *release*, size_t *extradatasize*, gfp_t *gfp*)
replace transfers with several transfers and register change with spi_message.resources

Parameters

struct spi_message * msg the spi_message we work upon

struct spi_transfer * xfer_first the first spi_transfer we want to replace

size_t remove number of transfers to remove

size_t insert the number of transfers we want to insert instead

spi_replaced_release_t release extra release code necessary in some circumstances

size_t extradatasize extra data to allocate (with alignment guarantees of struct **spi_transfer**)

gfp_t gfp gfp flags

Return

pointer to spi_replaced_transfers, PTR_ERR(…) in case of errors.

int **spi_split_transfers_maxsize**(struct *spi_controller* * *ctlr*, struct *spi_message* * *msg*, size_t *maxsize*, gfp_t *gfp*)
split spi transfers into multiple transfers when an individual transfer exceeds a certain size

Parameters

struct spi_controller * ctlr the **spi_controller** for this transfer

struct spi_message * msg the **spi_message** to transform

size_t maxsize the maximum when to apply this

gfp_t gfp GFP allocation flags

Return

status of transformation

int **spi_setup**(struct *spi_device* * *spi*)
setup SPI mode and clock rate

Parameters

struct spi_device * 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.

int **spi_async**(struct *spi_device* * *spi*, struct *spi_message* * *message*)
asynchronous SPI transfer

Parameters

struct spi_device * spi device with which data will be exchanged

struct spi_message * 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.

int **spi_async_locked**(struct *spi_device* * *spi*, struct *spi_message* * *message*)
version of spi_async with exclusive bus usage

Parameters

struct spi_device * spi device with which data will be exchanged

struct spi_message * 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.

int **spi_sync**(struct *spi_device* * *spi*, struct *spi_message* * *message*)
blocking/synchronous SPI data transfers

Parameters

`struct spi_device * spi` device with which data will be exchanged

`struct spi_message * 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.

int **`spi_sync_locked`**(struct *spi_device* * *spi*, struct *spi_message* * *message*)
    version of spi_sync with exclusive bus usage

Parameters

`struct spi_device * spi` device with which data will be exchanged

`struct spi_message * 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.

int **`spi_bus_lock`**(struct *spi_controller* * *ctlr*)
    obtain a lock for exclusive SPI bus usage

Parameters

`struct spi_controller * ctlr` 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.

int **spi_bus_unlock**(struct *spi_controller* * *ctlr*)
release the lock for exclusive SPI bus usage

Parameters

struct spi_controller * ctlr 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.

int **spi_write_then_read**(struct *spi_device* * *spi*, const void * *txbuf*, unsigned *n_tx*, void * *rxbuf*, unsigned *n_rx*)
SPI synchronous write followed by read

Parameters

struct spi_device * spi device with which data will be exchanged

const void * txbuf data to be written (need not be dma-safe)

unsigned n_tx size of txbuf, in bytes

void * rxbuf buffer into which data will be read (need not be dma-safe)

unsigned 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

SEVENTEEN

I^{2}C AND SMBUS SUBSYSTEM

I^{2}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 the master side of bus interactions and 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. 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.

int **i2c_master_recv**(const struct i2c_client * *client*, char * *buf*, int *count*)
issue a single I2C message in master receive mode

Parameters

const struct i2c_client * client Handle to slave device

char * buf Where to store data read from slave

int 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.

int **i2c_master_recv_dmasafe**(const struct i2c_client * *client*, char * *buf*, int *count*)
issue a single I2C message in master receive mode using a DMA safe buffer

Parameters

const struct i2c_client * client Handle to slave device

char * buf Where to store data read from slave, must be safe to use with DMA

int 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.

int **i2c_master_send**(const struct *i2c_client* * *client*, const char * *buf*, int *count*)
issue a single I2C message in master transmit mode

Parameters

const struct i2c_client * client Handle to slave device

const char * buf Data that will be written to the slave

int 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.

int **i2c_master_send_dmasafe**(const struct *i2c_client* * *client*, const char * *buf*, int *count*)
issue a single I2C message in master transmit mode using a DMA safe buffer

Parameters

const struct i2c_client * client Handle to slave device

const char * buf Data that will be written to the slave, must be safe to use with DMA

int 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.

struct **i2c_driver**
represent an I2C device driver

Definition

```
struct i2c_driver {
  unsigned int class;
  int (*attach_adapter)(struct i2c_adapter *) __deprecated;
  int (*probe)(struct i2c_client *, const struct i2c_device_id *);
  int (*remove)(struct i2c_client *);
  int (*probe_new)(struct i2c_client *);
  void (*shutdown)(struct i2c_client *);
  void (*alert)(struct i2c_client *, enum i2c_alert_protocol protocol, 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;
  bool disable_i2c_core_irq_mapping;
};
```

Members

class What kind of i2c device we instantiate (for detect)

attach_adapter Callback for bus addition (deprecated)

probe Callback for device binding - soon to be deprecated

remove Callback for device unbinding

probe_new New callback for device binding

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)

disable_i2c_core_irq_mapping Tell the i2c-core to not do irq-mapping

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.

struct **i2c_client**
represent an I2C slave device

Definition

```
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;
#if 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 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.

struct **i2c_board_info**
template for device creation

Definition

```
struct i2c_board_info {
  char type[I2C_NAME_SIZE];
  unsigned short  flags;
  unsigned short  addr;
  const char      *dev_name;
  void *platform_data;
  struct dev_archdata     *archdata;
  struct device_node *of_node;
  struct fwnode_handle *fwnode;
  const struct property_entry *properties;
  const struct resource *resources;
  unsigned int    num_resources;
  int irq;
};
```

Members

type chip type, to initialize i2c_client.name

flags to initialize i2c_client.flags

addr stored in i2c_client.addr

dev_name Overrides the default <busnr>-<addr> dev_name if set

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

properties additional device properties for the device

resources resources associated with the device

num_resources number of resources in the **resources** array

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.

I2C_BOARD_INFO(*dev_type*, *dev_addr*)
macro used to list an i2c device and its address

Parameters

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.

struct **i2c_algorithm**
represent I2C transfer method

Definition

```
struct i2c_algorithm {
  int (*master_xfer)(struct i2c_adapter *adap, struct i2c_msg *msgs, int num);
  int (*smbus_xfer) (struct i2c_adapter *adap, u16 addr,unsigned short flags, char read_write, u8 commar
  u32 (*functionality) (struct i2c_adapter *);
#if 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

Description

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.

struct **i2c_lock_operations**
represent I2C locking operations

Definition

```
struct i2c_lock_operations {
  void (*lock_bus)(struct i2c_adapter *, unsigned int flags);
  int (*trylock_bus)(struct i2c_adapter *, unsigned int flags);
  void (*unlock_bus)(struct i2c_adapter *, unsigned int flags);
};
```

Members

lock_bus Get exclusive access to an I2C bus segment

trylock_bus Try to get exclusive access to an I2C bus segment

unlock_bus Release exclusive access to an I2C bus segment

Description

The main operations are wrapped by i2c_lock_bus and i2c_unlock_bus.

struct **i2c_timings**
I2C timing information

Definition

```
struct i2c_timings {
  u32 bus_freq_hz;
  u32 scl_rise_ns;
  u32 scl_fall_ns;
  u32 scl_int_delay_ns;
  u32 sda_fall_ns;
};
```

Members

bus_freq_hz the bus frequency in Hz

scl_rise_ns time SCL signal takes to rise in ns; t(r) in the I2C specification

scl_fall_ns time SCL signal takes to fall in ns; t(f) in the I2C specification

scl_int_delay_ns time IP core additionally needs to setup SCL in ns

sda_fall_ns time SDA signal takes to fall in ns; t(f) in the I2C specification

struct **i2c_bus_recovery_info**
I2C bus recovery information

Definition

```
struct i2c_bus_recovery_info {
  int (*recover_bus)(struct i2c_adapter *adap);
  int (*get_scl)(struct i2c_adapter *adap);
  void (*set_scl)(struct i2c_adapter *adap, int val);
  int (*get_sda)(struct i2c_adapter *adap);
  void (*set_sda)(struct i2c_adapter *adap, int val);
  void (*prepare_recovery)(struct i2c_adapter *adap);
  void (*unprepare_recovery)(struct i2c_adapter *adap);
  struct gpio_desc *scl_gpiod;
  struct gpio_desc *sda_gpiod;
};
```

Members

recover_bus Recover routine. Either pass driver's `recover_bus()` routine, or `i2c_generic_scl_recovery()`.

get_scl This gets current value of SCL line. Mandatory for generic SCL recovery. Populated internally for generic GPIO recovery.

set_scl This sets/clears the SCL line. Mandatory for generic SCL recovery. Populated internally for generic GPIO recovery.

get_sda This gets current value of SDA line. Optional for generic SCL recovery. Populated internally, if sda_gpio is a valid GPIO, for generic GPIO recovery.

set_sda This sets/clears the SDA line. Optional for generic SCL recovery. Populated internally, if sda_gpio 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_gpiod gpiod of the SCL line. Only required for GPIO recovery.

sda_gpiod gpiod of the SDA line. Only required for GPIO recovery.

struct **i2c_adapter_quirks**
describe flaws of an i2c adapter

Definition

```
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

Description

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.

void **`i2c_lock_bus`**(struct i2c_adapter * *adapter*, unsigned int *flags*)
Get exclusive access to an I2C bus segment

Parameters

`struct i2c_adapter * adapter` Target I2C bus segment

`unsigned int flags` I2C_LOCK_ROOT_ADAPTER locks the root i2c adapter, I2C_LOCK_SEGMENT locks only this branch in the adapter tree

int **`i2c_trylock_bus`**(struct i2c_adapter * *adapter*, unsigned int *flags*)
Try to get exclusive access to an I2C bus segment

Parameters

`struct i2c_adapter * adapter` Target I2C bus segment

`unsigned int flags` I2C_LOCK_ROOT_ADAPTER tries to locks the root i2c adapter, I2C_LOCK_SEGMENT tries to lock only this branch in the adapter tree

Return

true if the I2C bus segment is locked, false otherwise

void **`i2c_unlock_bus`**(struct i2c_adapter * *adapter*, unsigned int *flags*)
Release exclusive access to an I2C bus segment

Parameters

`struct i2c_adapter * adapter` Target I2C bus segment

`unsigned int flags` I2C_LOCK_ROOT_ADAPTER unlocks the root i2c adapter, I2C_LOCK_SEGMENT unlocks only this branch in the adapter tree

bool **`i2c_check_quirks`**(struct i2c_adapter * *adap*, u64 *quirks*)
Function for checking the quirk flags in an i2c adapter

Parameters

`struct i2c_adapter * adap` i2c adapter

`u64 quirks` quirk flags

Return

true if the adapter has all the specified quirk flags, false if not

`module_i2c_driver`(*__i2c_driver*)
Helper macro for registering a modular I2C driver

Parameters

`__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()`*

`builtin_i2c_driver`(*__i2c_driver*)
Helper macro for registering a builtin I2C driver

Parameters

`__i2c_driver` i2c_driver struct

Description

Helper macro for I2C drivers which do not do anything special in their init. This eliminates a lot of boilerplate. Each driver may only use this macro once, and calling it replaces `device_initcall()`.

int **`i2c_register_board_info`**(int *busnum*, struct *i2c_board_info* const * *info*, unsigned *len*)
statically declare I2C devices

Parameters

`int busnum` identifies the bus to which these devices belong

`struct i2c_board_info const * info` vector of i2c device descriptors

`unsigned 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. Device properties are deep-copied though.

struct *i2c_client* * **`i2c_verify_client`**(struct *device* * *dev*)
return parameter as i2c_client, or NULL

Parameters

`struct device * 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.

struct *i2c_client* * **i2c_new_device**(struct i2c_adapter * *adap*, struct *i2c_board_info* const * *info*)
instantiate an i2c device

Parameters

struct i2c_adapter * adap the adapter managing the device

struct i2c_board_info const * 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.

void **i2c_unregister_device**(struct *i2c_client* * *client*)
reverse effect of *i2c_new_device()*

Parameters

struct i2c_client * client value returned from *i2c_new_device()*

Context

can sleep

struct *i2c_client* * **i2c_new_dummy**(struct i2c_adapter * *adapter*, u16 *address*)
return a new i2c device bound to a dummy driver

Parameters

struct i2c_adapter * adapter the adapter managing the device

u16 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.

struct *i2c_client* * **i2c_new_secondary_device**(struct *i2c_client* * *client*, const char * *name*, u16 *default_addr*)
Helper to get the instantiated secondary address and create the associated device

Parameters

struct i2c_client * client Handle to the primary client

const char * name Handle to specify which secondary address to get

u16 default_addr Used as a fallback if no secondary address was specified

Context

can sleep

Description

I2C clients can be composed of multiple I2C slaves bound together in a single component. The I2C client driver then binds to the master I2C slave and needs to create I2C dummy clients to communicate with all the other slaves.

This function creates and returns an I2C dummy client whose I2C address is retrieved from the platform firmware based on the given slave name. If no address is specified by the firmware default_addr is used.

On DT-based platforms the address is retrieved from the "reg" property entry cell whose "reg-names" value matches the slave name.

This returns the new i2c client, which should be saved for later use with *`i2c_unregister_device()`*; or NULL to indicate an error.

struct i2c_adapter * **`i2c_verify_adapter`**(struct *device* * *dev*)
    return parameter as i2c_adapter or NULL

Parameters

`struct device * 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.

int **`i2c_handle_smbus_host_notify`**(struct i2c_adapter * *adap*, unsigned short *addr*)
    Forward a Host Notify event to the correct I2C client.

Parameters

`struct i2c_adapter * adap` the adapter

`unsigned short addr` the I2C address of the notifying device

Context

can't sleep

Description

Helper function to be called from an I2C bus driver's interrupt handler. It will schedule the Host Notify IRQ.

int **`i2c_add_adapter`**(struct i2c_adapter * *adapter*)
    declare i2c adapter, use dynamic bus number

Parameters

`struct i2c_adapter * 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.

int **`i2c_add_numbered_adapter`**(struct i2c_adapter * *adap*)
    declare i2c adapter, use static bus number

Parameters

struct i2c_adapter * 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.

void **i2c_del_adapter**(struct i2c_adapter * *adap*)
    unregister I2C adapter

Parameters

struct i2c_adapter * 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**.

void **i2c_parse_fw_timings**(struct *device* * *dev*, struct *i2c_timings* * *t*, bool *use_defaults*)
    get I2C related timing parameters from firmware

Parameters

struct device * dev The device to scan for I2C timing properties

struct i2c_timings * t the i2c_timings struct to be filled with values

bool use_defaults bool to use sane defaults derived from the I2C specification when properties are not found, otherwise use 0

Description

Scan the device for the generic I2C properties describing timing parameters for the signal and fill the given struct with the results. If a property was not found and use_defaults was true, then maximum timings are assumed which are derived from the I2C specification. If use_defaults is not used, the results will be 0, so drivers can apply their own defaults later. The latter is mainly intended for avoiding regressions of existing drivers which want to switch to this function. New drivers almost always should use the defaults.

void **i2c_del_driver**(struct *i2c_driver* * *driver*)
    unregister I2C driver

Parameters

struct i2c_driver * driver the driver being unregistered

Context

can sleep

struct *i2c_client* * **i2c_use_client**(struct *i2c_client* * *client*)
increments the reference count of the i2c client structure

Parameters

struct i2c_client * 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.

void **i2c_release_client**(struct *i2c_client* * *client*)
release a use of the i2c client structure

Parameters

struct i2c_client * client the client being no longer referenced

Description

Must be called when a user of a client is finished with it.

int **__i2c_transfer**(struct i2c_adapter * *adap*, struct i2c_msg * *msgs*, int *num*)
unlocked flavor of i2c_transfer

Parameters

struct i2c_adapter * adap Handle to I2C bus

struct i2c_msg * msgs One or more messages to execute before STOP is issued to terminate the operation; each message begins with a START.

int 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.

int **i2c_transfer**(struct i2c_adapter * *adap*, struct i2c_msg * *msgs*, int *num*)
execute a single or combined I2C message

Parameters

struct i2c_adapter * adap Handle to I2C bus

struct i2c_msg * msgs One or more messages to execute before STOP is issued to terminate the operation; each message begins with a START.

int 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.

int **i2c_transfer_buffer_flags**(const struct *i2c_client* * *client*, char * *buf*, int *count*, u16 *flags*)
issue a single I2C message transferring data to/from a buffer

Parameters

const struct i2c_client * client Handle to slave device

char * buf Where the data is stored

int count How many bytes to transfer, must be less than 64k since msg.len is u16

u16 flags The flags to be used for the message, e.g. I2C_M_RD for reads

Description

Returns negative errno, or else the number of bytes transferred.

u8 * **i2c_get_dma_safe_msg_buf**(struct i2c_msg * *msg*, unsigned int *threshold*)
get a DMA safe buffer for the given i2c_msg

Parameters

struct i2c_msg * msg the message to be checked

unsigned int threshold the minimum number of bytes for which using DMA makes sense

Return

NULL if a DMA safe buffer was not obtained. Use msg->buf with PIO. Or a valid pointer to be used with DMA. After use, release it by calling *i2c_release_dma_safe_msg_buf()*.

This function must only be called from process context!

void **i2c_release_dma_safe_msg_buf**(struct i2c_msg * *msg*, u8 * *buf*)
release DMA safe buffer and sync with i2c_msg

Parameters

struct i2c_msg * msg the message to be synced with

u8 * buf the buffer obtained from *i2c_get_dma_safe_msg_buf()*. May be NULL.

s32 **i2c_smbus_read_byte**(const struct *i2c_client* * *client*)
SMBus "receive byte" protocol

Parameters

const struct i2c_client * client Handle to slave device

Description

This executes the SMBus "receive byte" protocol, returning negative errno else the byte received from the device.

s32 **i2c_smbus_write_byte**(const struct *i2c_client* * *client*, u8 *value*)
SMBus "send byte" protocol

Parameters

const struct i2c_client * client Handle to slave device

u8 value Byte to be sent

Description

This executes the SMBus "send byte" protocol, returning negative errno else zero on success.

s32 **i2c_smbus_read_byte_data**(const struct *i2c_client* * *client*, u8 *command*)
SMBus "read byte" protocol

Parameters

const struct i2c_client * client Handle to slave device

u8 command Byte interpreted by slave

Description

This executes the SMBus "read byte" protocol, returning negative errno else a data byte received from the device.

s32 **i2c_smbus_write_byte_data**(const struct *i2c_client* * *client*, u8 *command*, u8 *value*)
SMBus "write byte" protocol

Parameters

const struct i2c_client * client Handle to slave device

u8 command Byte interpreted by slave

u8 value Byte being written

Description

This executes the SMBus "write byte" protocol, returning negative errno else zero on success.

s32 **i2c_smbus_read_word_data**(const struct *i2c_client* * *client*, u8 *command*)
SMBus "read word" protocol

Parameters

const struct i2c_client * client Handle to slave device

u8 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.

s32 **i2c_smbus_write_word_data**(const struct *i2c_client* * *client*, u8 *command*, u16 *value*)
SMBus "write word" protocol

Parameters

const struct i2c_client * client Handle to slave device

u8 command Byte interpreted by slave

u16 value 16-bit "word" being written

Description

This executes the SMBus "write word" protocol, returning negative errno else zero on success.

s32 **i2c_smbus_read_block_data**(const struct *i2c_client* * *client*, u8 *command*, u8 * *values*)
SMBus "block read" protocol

Parameters

const struct i2c_client * client Handle to slave device

u8 command Byte interpreted by slave

u8 * 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.

s32 **i2c_smbus_write_block_data**(const struct *i2c_client* * *client*, u8 *command*, u8 *length*, const u8 * *values*)
SMBus "block write" protocol

Parameters

const struct i2c_client * client Handle to slave device

u8 command Byte interpreted by slave

u8 length Size of data block; SMBus allows at most 32 bytes

const u8 * values Byte array which will be written.

Description

This executes the SMBus “block write” protocol, returning negative errno else zero on success.

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*)
execute SMBus protocol operations

Parameters

struct i2c_adapter * adapter Handle to I2C bus

u16 addr Address of SMBus slave on that bus

unsigned short flags I2C_CLIENT_* flags (usually zero or I2C_CLIENT_PEC)

char read_write I2C_SMBUS_READ or I2C_SMBUS_WRITE

u8 command Byte interpreted by slave, for protocols which use such bytes

int protocol SMBus protocol operation to execute, such as I2C_SMBUS_PROC_CALL

union i2c_smbus_data * data Data to be read or written

Description

This executes an SMBus protocol operation, and returns a negative errno code else zero on success.

s32 **i2c_smbus_read_i2c_block_data_or_emulated**(const struct *i2c_client* * *client*, u8 *command*, u8 *length*, u8 * *values*)
read block or emulate

Parameters

const struct i2c_client * client Handle to slave device

u8 command Byte interpreted by slave

u8 length Size of data block; SMBus allows at most I2C_SMBUS_BLOCK_MAX bytes

u8 * values 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.

struct *i2c_client* * **i2c_setup_smbus_alert**(struct i2c_adapter * *adapter*, struct i2c_smbus_alert_setup * *setup*)
Setup SMBus alert support

Parameters

struct i2c_adapter * adapter the target adapter

struct i2c_smbus_alert_setup * setup setup data for the SMBus alert handler

Context

can sleep

Description

Setup handling of the SMBus alert protocol on a given I2C bus segment.

Handling can be done either through our IRQ handler, or by the adapter (from its handler, periodic polling, or whatever).

NOTE that if we manage the IRQ, we *MUST* know if it's level or edge triggered in order to hand it to the workqueue correctly. If triggering the alert seems to wedge the system, you probably should have said it's level triggered.

This returns the ara client, which should be saved for later use with `i2c_handle_smbus_alert()` and ultimately *`i2c_unregister_device()`*; or NULL to indicate an error.

CHAPTER

EIGHTEEN

HIGH SPEED SYNCHRONOUS SERIAL INTERFACE (HSI)

Introduction

High Speed Syncronous Interface (HSI) is a fullduplex, low latency protocol, that is optimized for die-level interconnect between an Application Processor and a Baseband chipset. It has been specified by the MIPI alliance in 2003 and implemented by multiple vendors since then.

The HSI interface supports full duplex communication over multiple channels (typically 8) and is capable of reaching speeds up to 200 Mbit/s.

The serial protocol uses two signals, DATA and FLAG as combined data and clock signals and an additional READY signal for flow control. An additional WAKE signal can be used to wakeup the chips from standby modes. The signals are commonly prefixed by AC for signals going from the application die to the cellular die and CA for signals going the other way around.

```
+------------+                               +---------------+
|  Cellular  |                               |  Application  |
|    Die     |                               |      Die      |
|            | - - - - - - CAWAKE - - - - - - >|               |
|           T|------------ CADATA ------------>|R              |
|           X|------------ CAFLAG ------------>|X              |
|            |<----------- ACREADY ------------|               |
|            |                               |               |
|            |                               |               |
|            |< - - - - -  ACWAKE - - - - - - -|               |
|           R|<----------- ACDATA -------------|T              |
|           X|<----------- ACFLAG -------------|X              |
|            |------------ CAREADY ----------->|               |
|            |                               |               |
|            |                               |               |
+------------+                               +---------------+
```

HSI Subsystem in Linux

In the Linux kernel the hsi subsystem is supposed to be used for HSI devices. The hsi subsystem contains drivers for hsi controllers including support for multi-port controllers and provides a generic API for using the HSI ports.

It also contains HSI client drivers, which make use of the generic API to implement a protocol used on the HSI interface. These client drivers can use an arbitrary number of channels.

hsi-char Device

Each port automatically registers a generic client driver called hsi_char, which provides a charecter device for userspace representing the HSI port. It can be used to communicate via HSI from userspace. Userspace may configure the hsi_char device using the following ioctl commands:

HSC_RESET flush the HSI port

HSC_SET_PM enable or disable the client.

HSC_SEND_BREAK send break

HSC_SET_RX set RX configuration

HSC_GET_RX get RX configuration

HSC_SET_TX set TX configuration

HSC_GET_TX get TX configuration

The kernel HSI API

struct **hsi_channel**
channel resource used by the hsi clients

Definition

```
struct hsi_channel {
  unsigned int    id;
  const char      *name;
};
```

Members

id Channel number

name Channel name

struct **hsi_config**
Configuration for RX/TX HSI modules

Definition

```
struct hsi_config {
  unsigned int            mode;
  struct hsi_channel      *channels;
  unsigned int            num_channels;
  unsigned int            num_hw_channels;
  unsigned int            speed;
  union {
    unsigned int    flow;
    unsigned int    arb_mode;
  };
};
```

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

flow RX flow type (SYNCHRONIZED or PIPELINE)

arb_mode Arbitration mode for TX frame (Round robin, priority)

struct **hsi_board_info**
HSI client board info

Definition

```
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

struct **hsi_client**
HSI client attached to an HSI port

Definition

```
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

struct **hsi_client_driver**
Driver associated to an HSI client

Definition

```
struct hsi_client_driver {
  struct device_driver    driver;
};
```

Members

driver Driver model representation of the driver

struct **hsi_msg**
HSI message descriptor

Definition

```
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.

struct **hsi_port**
HSI port device

Definition

```
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 blocking_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.

struct **hsi_controller**
HSI controller device

Definition

```
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

unsigned int **hsi_id**(struct *hsi_client* * *cl*)
Get HSI controller ID associated to a client

Parameters

struct hsi_client * cl Pointer to a HSI client

Description

Return the controller id where the client is attached to

unsigned int **hsi_port_id**(struct *hsi_client* * *cl*)
Gets the port number a client is attached to

Parameters

struct hsi_client * cl Pointer to HSI client

Description

Return the port number associated to the client

int **hsi_setup**(struct *hsi_client* * *cl*)
Configure the client's port

Parameters

struct hsi_client * 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

int **hsi_flush**(struct *hsi_client* * *cl*)
Flush all pending transactions on the client's port

Parameters

struct hsi_client * 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

int **hsi_async_read**(struct *hsi_client* * *cl*, struct *hsi_msg* * *msg*)
Submit a read transfer

Parameters

struct hsi_client * cl Pointer to the HSI client

struct hsi_msg * msg HSI message descriptor of the transfer

Description

Return -errno on failure, 0 on success

int **hsi_async_write**(struct *hsi_client* * *cl*, struct *hsi_msg* * *msg*)
Submit a write transfer

Parameters

struct hsi_client * cl Pointer to the HSI client

struct hsi_msg * msg HSI message descriptor of the transfer

Description

Return -errno on failure, 0 on success

int **hsi_start_tx**(struct *hsi_client* * *cl*)
Signal the port that the client wants to start a TX

Parameters

struct hsi_client * cl Pointer to the HSI client

Description

Return -errno on failure, 0 on success

int **hsi_stop_tx**(struct *hsi_client* * *cl*)
Signal the port that the client no longer wants to transmit

Parameters

struct hsi_client * cl Pointer to the HSI client

Description

Return -errno on failure, 0 on success

void **hsi_port_unregister_clients**(struct *hsi_port* * *port*)
Unregister an HSI port

Parameters

struct hsi_port * port The HSI port to unregister

void **hsi_unregister_controller**(struct *hsi_controller* * *hsi*)
Unregister an HSI controller

Parameters

struct hsi_controller * hsi The HSI controller to register

int **hsi_register_controller**(struct *hsi_controller* * *hsi*)
Register an HSI controller and its ports

Parameters

struct hsi_controller * hsi The HSI controller to register

Description

Returns -errno on failure, 0 on success.

int **hsi_register_client_driver**(struct *hsi_client_driver* * *drv*)
Register an HSI client to the HSI bus

Parameters

struct hsi_client_driver * drv HSI client driver to register

Description

Returns -errno on failure, 0 on success.

void **hsi_put_controller**(struct *hsi_controller* * *hsi*)
Free an HSI controller

Parameters

struct hsi_controller * 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.

struct *hsi_controller* * **hsi_alloc_controller**(unsigned int *n_ports*, gfp_t *flags*)
Allocate an HSI controller and its ports

Parameters

unsigned int n_ports Number of ports on the HSI controller

gfp_t flags Kernel allocation flags

Description

Return NULL on failure or a pointer to an hsi_controller on success.

void **hsi_free_msg**(struct *hsi_msg* * *msg*)
Free an HSI message

Parameters

struct hsi_msg * msg Pointer to the HSI message

Description

Client is responsible to free the buffers pointed by the scatterlists.

struct *hsi_msg* * **hsi_alloc_msg**(unsigned int *nents*, gfp_t *flags*)
Allocate an HSI message

Parameters

unsigned int nents Number of memory entries

gfp_t 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.

int **hsi_async**(struct *hsi_client* * *cl*, struct *hsi_msg* * *msg*)
   Submit an HSI transfer to the controller

Parameters

struct hsi_client * cl HSI client sending the transfer

struct hsi_msg * 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

int **hsi_claim_port**(struct *hsi_client* * *cl*, unsigned int *share*)
   Claim the HSI client's port

Parameters

struct hsi_client * cl HSI client that wants to claim its port

unsigned int share Flag to indicate if the client wants to share the port or not.

Description

Returns -errno on failure, 0 on success.

void **hsi_release_port**(struct *hsi_client* * *cl*)
   Release the HSI client's port

Parameters

struct hsi_client * cl HSI client which previously claimed its port

int **hsi_register_port_event**(struct *hsi_client* * *cl*, void (*handler) (struct *hsi_client* *, unsigned *long*)
   Register a client to receive port events

Parameters

struct hsi_client * cl HSI client that wants to receive port events

void (*)(struct hsi_client *, unsigned long) 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.

int **hsi_unregister_port_event**(struct *hsi_client* * *cl*)
   Stop receiving port events for a client

Parameters

struct hsi_client * 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.

int **hsi_event**(struct *hsi_port* * *port*, unsigned long *event*)
    Notifies clients about port events

Parameters

struct hsi_port * port Port where the event occurred

unsigned long 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.

int **hsi_get_channel_id_by_name**(struct *hsi_client* * *cl*, char * *name*)
    acquire channel id by channel name

Parameters

struct hsi_client * cl HSI client, which uses the channel

char * 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

NINETEEN

ERROR DETECTION AND CORRECTION (EDAC) DEVICES

Main Concepts used at the EDAC subsystem

There are several things to be aware of that aren't at all obvious, like *sockets, *socket sets*, *banks*, *rows*, *chip-select rows*, *channels*, etc...

These are some of the many terms that are thrown about that don't always mean what people think they mean (Inconceivable!). In the interest of creating a common ground for discussion, terms and their definitions will be established.

- Memory devices

The individual DRAM chips on a memory stick. These devices commonly output 4 and 8 bits each (x4, x8). Grouping several of these in parallel provides the number of bits that the memory controller expects: typically 72 bits, in order to provide 64 bits + 8 bits of ECC data.

- Memory Stick

A printed circuit board that aggregates multiple memory devices in parallel. In general, this is the Field Replaceable Unit (FRU) which gets replaced, in the case of excessive errors. Most often it is also called DIMM (Dual Inline Memory Module).

- Memory Socket

A physical connector on the motherboard that accepts a single memory stick. Also called as "slot" on several datasheets.

- Channel

A memory controller channel, responsible to communicate with a group of DIMMs. Each channel has its own independent control (command) and data bus, and can be used independently or grouped with other channels.

- Branch

It is typically the highest hierarchy on a Fully-Buffered DIMM memory controller. Typically, it contains two channels. Two channels at the same branch can be used in single mode or in lockstep mode. When lockstep is enabled, the cacheline is doubled, but it generally brings some performance penalty. Also, it is generally not possible to point to just one memory stick when an error occurs, as the error correction code is calculated using two DIMMs instead of one. Due to that, it is capable of correcting more errors than on single mode.

- Single-channel

The data accessed by the memory controller is contained into one dimm only. E. g. if the data is 64 bits-wide, the data flows to the CPU using one 64 bits parallel access. Typically used with SDR, DDR, DDR2 and DDR3 memories. FB-DIMM and RAMBUS use a different concept for channel, so this concept doesn't apply there.

- Double-channel

The data size accessed by the memory controller is interlaced into two dimms, accessed at the same time. E. g. if the DIMM is 64 bits-wide (72 bits with ECC), the data flows to the CPU using a 128 bits parallel access.

- Chip-select row

This is the name of the DRAM signal used to select the DRAM ranks to be accessed. Common chip-select rows for single channel are 64 bits, for dual channel 128 bits. It may not be visible by the memory controller, as some DIMM types have a memory buffer that can hide direct access to it from the Memory Controller.

- Single-Ranked stick

A Single-ranked stick has 1 chip-select row of memory. Motherboards commonly drive two chip-select pins to a memory stick. A single-ranked stick, will occupy only one of those rows. The other will be unused.

- Double-Ranked stick

A double-ranked stick has two chip-select rows which access different sets of memory devices. The two rows cannot be accessed concurrently.

- Double-sided stick

DEPRECATED TERM, see *Double-Ranked stick*.

A double-sided stick has two chip-select rows which access different sets of memory devices. The two rows cannot be accessed concurrently. “Double-sided” is irrespective of the memory devices being mounted on both sides of the memory stick.

- Socket set

All of the memory sticks that are required for a single memory access or all of the memory sticks spanned by a chip-select row. A single socket set has two chip-select rows and if double-sided sticks are used these will occupy those chip-select rows.

- Bank

This term is avoided because it is unclear when needing to distinguish between chip-select rows and socket sets.

Memory Controllers

Most of the EDAC core is focused on doing Memory Controller error detection. The *edac_mc_alloc()*. It uses internally the struct `mem_ctl_info` to describe the memory controllers, with is an opaque struct for the EDAC drivers. Only the EDAC core is allowed to touch it.

enum **dev_type**
describe the type of memory DRAM chips used at the stick

Constants

DEV_UNKNOWN Can’t be determined, or MC doesn’t support detect it

DEV_X1 1 bit for data

DEV_X2 2 bits for data

DEV_X4 4 bits for data

DEV_X8 8 bits for data

DEV_X16 16 bits for data

DEV_X32 32 bits for data

DEV_X64 64 bits for data

Description

Typical values are x4 and x8.

enum **hw_event_mc_err_type**
type of the detected error

Constants

HW_EVENT_ERR_CORRECTED Corrected Error - Indicates that an ECC corrected error was detected

HW_EVENT_ERR_UNCORRECTED Uncorrected Error - Indicates an error that can't be corrected by ECC, but it is not fatal (maybe it is on an unused memory area, or the memory controller could recover from it for example, by re-trying the operation).

HW_EVENT_ERR_DEFERRED Deferred Error - Indicates an uncorrectable error whose handling is not urgent. This could be due to hardware data poisoning where the system can continue operation until the poisoned data is consumed. Preemptive measures may also be taken, e.g. offlining pages, etc.

HW_EVENT_ERR_FATAL Fatal Error - Uncorrected error that could not be recovered.

HW_EVENT_ERR_INFO Informational - The CPER spec defines a forth type of error: informational logs.

enum **mem_type**
memory types. For a more detailed reference, please see http://en.wikipedia.org/wiki/DRAM

Constants

MEM_EMPTY Empty csrow

MEM_RESERVED Reserved csrow type

MEM_UNKNOWN Unknown csrow type

MEM_FPM FPM - Fast Page Mode, used on systems up to 1995.

MEM_EDO EDO - Extended data out, used on systems up to 1998.

MEM_BEDO BEDO - Burst Extended data out, an EDO variant.

MEM_SDR SDR - Single data rate SDRAM http://en.wikipedia.org/wiki/Synchronous_dynamic_random-access_memory They use 3 pins for chip select: Pins 0 and 2 are for rank 0; pins 1 and 3 are for rank 1, if the memory is dual-rank.

MEM_RDR Registered SDR SDRAM

MEM_DDR Double data rate SDRAM http://en.wikipedia.org/wiki/DDR_SDRAM

MEM_RDDR Registered Double data rate SDRAM This is a variant of the DDR memories. A registered memory has a buffer inside it, hiding part of the memory details to the memory controller.

MEM_RMBS Rambus DRAM, used on a few Pentium III/IV controllers.

MEM_DDR2 DDR2 RAM, as described at JEDEC JESD79-2F. Those memories are labeled as "PC2-" instead of "PC" to differentiate from DDR.

MEM_FB_DDR2 Fully-Buffered DDR2, as described at JEDEC Std No. 205 and JESD206. Those memories are accessed per DIMM slot, and not by a chip select signal.

MEM_RDDR2 Registered DDR2 RAM This is a variant of the DDR2 memories.

MEM_XDR Rambus XDR It is an evolution of the original RAMBUS memories, created to compete with DDR2. Weren't used on any x86 arch, but cell_edac PPC memory controller uses it.

MEM_DDR3 DDR3 RAM

MEM_RDDR3 Registered DDR3 RAM This is a variant of the DDR3 memories.

MEM_LRDDR3 Load-Reduced DDR3 memory.

MEM_DDR4 Unbuffered DDR4 RAM

MEM_RDDR4 Registered DDR4 RAM This is a variant of the DDR4 memories.

`MEM_LRDDR4` Load-Reduced DDR4 memory.

enum **`edac_type`**
type - Error Detection and Correction capabilities and mode

Constants

`EDAC_UNKNOWN` Unknown if ECC is available

`EDAC_NONE` Doesn't support ECC

`EDAC_RESERVED` Reserved ECC type

`EDAC_PARITY` Detects parity errors

`EDAC_EC` Error Checking - no correction

`EDAC_SECDED` Single bit error correction, Double detection

`EDAC_S2ECD2ED` Chipkill x2 devices - do these exist?

`EDAC_S4ECD4ED` Chipkill x4 devices

`EDAC_S8ECD8ED` Chipkill x8 devices

`EDAC_S16ECD16ED` Chipkill x16 devices

enum **`scrub_type`**
scrubbing capabilities

Constants

`SCRUB_UNKNOWN` Unknown if scrubber is available

`SCRUB_NONE` No scrubber

`SCRUB_SW_PROG` SW progressive (sequential) scrubbing

`SCRUB_SW_SRC` Software scrub only errors

`SCRUB_SW_PROG_SRC` Progressive software scrub from an error

`SCRUB_SW_TUNABLE` Software scrub frequency is tunable

`SCRUB_HW_PROG` HW progressive (sequential) scrubbing

`SCRUB_HW_SRC` Hardware scrub only errors

`SCRUB_HW_PROG_SRC` Progressive hardware scrub from an error

`SCRUB_HW_TUNABLE` Hardware scrub frequency is tunable

enum **`edac_mc_layer_type`**
memory controller hierarchy layer

Constants

`EDAC_MC_LAYER_BRANCH` memory layer is named "branch"

`EDAC_MC_LAYER_CHANNEL` memory layer is named "channel"

`EDAC_MC_LAYER_SLOT` memory layer is named "slot"

`EDAC_MC_LAYER_CHIP_SELECT` memory layer is named "chip select"

`EDAC_MC_LAYER_ALL_MEM` memory layout is unknown. All memory is mapped as a single memory area. This is used when retrieving errors from a firmware driven driver.

Description

This enum is used by the drivers to tell edac_mc_sysfs what name should be used when describing a memory stick location.

struct **`edac_mc_layer`**
describes the memory controller hierarchy

Definition

```
struct edac_mc_layer {
  enum edac_mc_layer_type type;
  unsigned size;
  bool is_virt_csrow;
};
```

Members

type layer type

size number of components per layer. For example, if the channel layer has two channels, size = 2

is_virt_csrow This layer is part of the "csrow" when old API compatibility mode is enabled. Otherwise, it is a channel

EDAC_DIMM_OFF(*layers*, *nlayers*, *layer0*, *layer1*, *layer2*)
Macro responsible to get a pointer offset inside a pointer array for the element given by [layer0,layer1,layer2] position

Parameters

layers a struct edac_mc_layer array, describing how many elements were allocated for each layer

nlayers Number of layers at the **layers** array

layer0 layer0 position

layer1 layer1 position. Unused if n_layers < 2

layer2 layer2 position. Unused if n_layers < 3

Description

For 1 layer, this macro returns "var[layer0] - var";

For 2 layers, this macro is similar to allocate a bi-dimensional array and to return "var[layer0][layer1] - var";

For 3 layers, this macro is similar to allocate a tri-dimensional array and to return "var[layer0][layer1][layer2] - var".

A loop could be used here to make it more generic, but, as we only have 3 layers, this is a little faster.

By design, layers can never be 0 or more than 3. If that ever happens, a NULL is returned, causing an OOPS during the memory allocation routine, with would point to the developer that he's doing something wrong.

EDAC_DIMM_PTR(*layers*, *var*, *nlayers*, *layer0*, *layer1*, *layer2*)
Macro responsible to get a pointer inside a pointer array for the element given by [layer0,layer1,layer2] position

Parameters

layers a struct edac_mc_layer array, describing how many elements were allocated for each layer

var name of the var where we want to get the pointer (like mci->dimms)

nlayers Number of layers at the **layers** array

layer0 layer0 position

layer1 layer1 position. Unused if n_layers < 2

layer2 layer2 position. Unused if n_layers < 3

Description

For 1 layer, this macro returns "var[layer0]";

For 2 layers, this macro is similar to allocate a bi-dimensional array and to return "var[layer0][layer1]";

For 3 layers, this macro is similar to allocate a tri-dimensional array and to return "var[layer0][layer1][layer2]";

struct **rank_info**
contains the information for one DIMM rank

Definition

```
struct rank_info {
  int chan_idx;
  struct csrow_info *csrow;
  struct dimm_info *dimm;
  u32 ce_count;
};
```

Members

chan_idx channel number where the rank is (typically, 0 or 1)

csrow A pointer to the chip select row structure (the parent structure). The location of the rank is given by the (csrow->csrow_idx, chan_idx) vector.

dimm A pointer to the DIMM structure, where the DIMM label information is stored.

ce_count number of correctable errors for this rank

Description

FIXME: Currently, the EDAC core model will assume one DIMM per rank. This is a bad assumption, but it makes this patch easier. Later patches in this series will fix this issue.

struct **edac_raw_error_desc**
Raw error report structure

Definition

```
struct edac_raw_error_desc {
  char location[LOCATION_SIZE];
  char label[(EDAC_MC_LABEL_LEN + 1 + sizeof(OTHER_LABEL)) * EDAC_MAX_LABELS];
  long grain;
  u16 error_count;
  int top_layer;
  int mid_layer;
  int low_layer;
  unsigned long page_frame_number;
  unsigned long offset_in_page;
  unsigned long syndrome;
  const char *msg;
  const char *other_detail;
  bool enable_per_layer_report;
};
```

Members

location location of the error

label label of the affected DIMM(s)

grain minimum granularity for an error report, in bytes

error_count number of errors of the same type

top_layer top layer of the error (layer[0])

mid_layer middle layer of the error (layer[1])

low_layer low layer of the error (layer[2])

page_frame_number page where the error happened

offset_in_page page offset

syndrome syndrome of the error (or 0 if unknown or if the syndrome is not applicable)

msg error message

other_detail other driver-specific detail about the error

enable_per_layer_report if false, the error affects all layers (typically, a memory controller error)

struct mem_ctl_info * **edac_mc_alloc**(unsigned *mc_num*, unsigned *n_layers*, struct *edac_mc_layer * layers*, unsigned *sz_pvt*)

Allocate and partially fill a struct `mem_ctl_info`.

Parameters

unsigned mc_num Memory controller number

unsigned n_layers Number of MC hierarchy layers

struct edac_mc_layer * layers Describes each layer as seen by the Memory Controller

unsigned sz_pvt size of private storage needed

Description

Everything is kmalloc'ed as one big chunk - more efficient. Only can be used if all structures have the same lifetime - otherwise you have to allocate and initialize your own structures.

Use `edac_mc_free()` to free mc structures allocated by this function.

Note:

drivers handle multi-rank memories in different ways: in some drivers, one multi-rank memory stick is mapped as one entry, while, in others, a single multi-rank memory stick would be mapped into several entries. Currently, this function will allocate multiple struct dimm_info on such scenarios, as grouping the multiple ranks require drivers change.

Return

On success, return a pointer to struct mem_ctl_info pointer; NULL otherwise

const char * **edac_get_owner**(void)

Return the owner's mod_name of EDAC MC

Parameters

void no arguments

Return

Pointer to mod_name string when EDAC MC is owned. NULL otherwise.

void **edac_mc_free**(struct mem_ctl_info * *mci*)

Frees a previously allocated **mci** structure

Parameters

struct mem_ctl_info * mci pointer to a struct mem_ctl_info structure

bool **edac_has_mcs**(void)

Check if any MCs have been allocated.

Parameters

void no arguments

Return

True if MC instances have been registered successfully. False otherwise.

struct mem_ctl_info * **edac_mc_find**(int *idx*)
Search for a mem_ctl_info structure whose index is **idx**.

Parameters

int idx index to be seek

Description

If found, return a pointer to the structure. Else return NULL.

struct mem_ctl_info * **find_mci_by_dev**(struct *device* * *dev*)
Scan list of controllers looking for the one that manages the **dev** device.

Parameters

struct device * dev pointer to a struct device related with the MCI

Return

on success, returns a pointer to struct `mem_ctl_info`; NULL otherwise.

struct mem_ctl_info * **edac_mc_del_mc**(struct *device* * *dev*)
Remove sysfs entries for mci structure associated with **dev** and remove mci structure from global list.

Parameters

struct device * dev Pointer to struct *device* representing mci structure to remove.

Return

pointer to removed mci structure, or NULL if device not found.

int **edac_mc_find_csrow_by_page**(struct mem_ctl_info * *mci*, unsigned long *page*)
Ancillary routine to identify what csrow contains a memory page.

Parameters

struct mem_ctl_info * mci pointer to a struct mem_ctl_info structure

unsigned long page memory page to find

Return

on success, returns the csrow. -1 if not found.

void **edac_raw_mc_handle_error**(const enum *hw_event_mc_err_type* *type*, struct mem_ctl_info * *mci*, struct *edac_raw_error_desc* * *e*)
Reports a memory event to userspace without doing anything to discover the error location.

Parameters

const enum hw_event_mc_err_type type severity of the error (CE/UE/Fatal)

struct mem_ctl_info * mci a struct mem_ctl_info pointer

struct edac_raw_error_desc * e error description

Description

This raw function is used internally by *edac_mc_handle_error()*. It should only be called directly when the hardware error come directly from BIOS, like in the case of APEI GHES driver.

void **edac_mc_handle_error**(const enum *hw_event_mc_err_type* *type*, struct mem_ctl_info * *mci*, const u16 *error_count*, const unsigned long *page_frame_number*, const unsigned long *offset_in_page*, const unsigned long *syndrome*, const int *top_layer*, const int *mid_layer*, const int *low_layer*, const char * *msg*, const char * *other_detail*)
Reports a memory event to userspace.

Parameters

const enum hw_event_mc_err_type type severity of the error (CE/UE/Fatal)

struct mem_ctl_info * mci a struct mem_ctl_info pointer

const u16 error_count Number of errors of the same type

const unsigned long page_frame_number mem page where the error occurred

const unsigned long offset_in_page offset of the error inside the page

const unsigned long syndrome ECC syndrome

const int top_layer Memory layer[0] position

const int mid_layer Memory layer[1] position

const int low_layer Memory layer[2] position

const char * msg Message meaningful to the end users that explains the event

const char * other_detail Technical details about the event that may help hardware manufacturers and EDAC developers to analyse the event

PCI Controllers

The EDAC subsystem provides a mechanism to handle PCI controllers by calling the *edac_pci_alloc_ctl_info()*. It will use the struct `edac_pci_ctl_info` to describe the PCI controllers.

struct edac_pci_ctl_info * **edac_pci_alloc_ctl_info**(unsigned int *sz_pvt*, const char * *edac_pci_name*)

Parameters

unsigned int sz_pvt size of the private info at struct `edac_pci_ctl_info`

const char * edac_pci_name name of the PCI device

Description

The `alloc()` function for the 'edac_pci' control info structure.

The chip driver will allocate one of these for each edac_pci it is going to control/register with the EDAC CORE.

Return

a pointer to struct `edac_pci_ctl_info` on success; `NULL` otherwise.

void **edac_pci_free_ctl_info**(struct edac_pci_ctl_info * *pci*)

Parameters

struct edac_pci_ctl_info * pci pointer to struct `edac_pci_ctl_info`

Description

Last action on the pci control structure.

Calls the remove sysfs information, which will unregister this control struct's kobj. When that kobj's ref count goes to zero, its release function will be call and then `kfree()` the memory.

int **edac_pci_alloc_index**(void)

Parameters

void no arguments

Return

allocated index number

int **edac_pci_add_device**(struct edac_pci_ctl_info * *pci*, int *edac_idx*)

Parameters

struct edac_pci_ctl_info * pci pointer to the edac_device structure to be added to the list

int edac_idx A unique numeric identifier to be assigned to the 'edac_pci' structure.

Description

edac_pci global list and create sysfs entries associated with edac_pci structure.

Return

0 on Success, or an error code on failure

struct edac_pci_ctl_info * **edac_pci_del_device**(struct *device* * *dev*)

Parameters

struct device * dev Pointer to 'struct device' representing edac_pci structure to remove

Description

Remove sysfs entries for specified edac_pci structure and then remove edac_pci structure from global list

Return

Pointer to removed edac_pci structure, or NULL if device not found

struct edac_pci_ctl_info * **edac_pci_create_generic_ctl**(struct *device* * *dev*, const char * *mod_name*)

Parameters

struct device * dev pointer to struct *device*;

const char * mod_name name of the PCI device

Description

A generic constructor for a PCI parity polling device Some systems have more than one domain of PCI busses. For systems with one domain, then this API will provide for a generic poller.

This routine calls the *edac_pci_alloc_ctl_info()* for the generic device, with default values

Return

Pointer to struct edac_pci_ctl_info on success, NULL on failure.

void **edac_pci_release_generic_ctl**(struct edac_pci_ctl_info * *pci*)

Parameters

struct edac_pci_ctl_info * pci pointer to struct edac_pci_ctl_info

Description

The release function of a generic EDAC PCI polling device

int **edac_pci_create_sysfs**(struct edac_pci_ctl_info * *pci*)

Parameters

struct edac_pci_ctl_info * pci pointer to struct edac_pci_ctl_info

Description

Create the controls/attributes for the specified EDAC PCI device

void **edac_pci_remove_sysfs**(struct edac_pci_ctl_info * *pci*)

Parameters

struct edac_pci_ctl_info * pci pointer to struct edac_pci_ctl_info

Description

remove the controls and attributes for this EDAC PCI device

EDAC Blocks

The EDAC subsystem also provides a generic mechanism to report errors on other parts of the hardware via `edac_device_alloc_ctl_info()` function.

The structures `edac_dev_sysfs_block_attribute`, `edac_device_block`, `edac_device_instance` and `edac_device_ctl_info` provide a generic or abstract 'edac_device' representation at sysfs.

This set of structures and the code that implements the APIs for the same, provide for registering EDAC type devices which are NOT standard memory or PCI, like:

- CPU caches (L1 and L2)
- DMA engines
- Core CPU switches
- Fabric switch units
- PCIe interface controllers
- other EDAC/ECC type devices that can be monitored for errors, etc.

It allows for a 2 level set of hierarchy.

For example, a cache could be composed of L1, L2 and L3 levels of cache. Each CPU core would have its own L1 cache, while sharing L2 and maybe L3 caches. On such case, those can be represented via the following sysfs nodes:

```
/sys/devices/system/edac/..

pci/            <existing pci directory (if available)>
mc/             <existing memory device directory>
cpu/cpu0/..     <L1 and L2 block directory>
        /L1-cache/ce_count
                 /ue_count
        /L2-cache/ce_count
                 /ue_count
cpu/cpu1/..     <L1 and L2 block directory>
        /L1-cache/ce_count
                 /ue_count
        /L2-cache/ce_count
                 /ue_count
...

the L1 and L2 directories would be "edac_device_block's"
```

int **`edac_device_add_device`**(struct edac_device_ctl_info * *edac_dev*)

Parameters

`struct edac_device_ctl_info * edac_dev` pointer to edac_device structure to be added to the list 'edac_device' structure.

Description

edac_device global list and create sysfs entries associated with edac_device structure.

Return

0 on Success, or an error code on failure

struct edac_device_ctl_info * **`edac_device_del_device`**(struct *device* * *dev*)

Parameters

struct device * dev Pointer to struct *device* representing the edac device structure to remove.

Description

Remove sysfs entries for specified edac_device structure and then remove edac_device structure from global list

Return

Pointer to removed edac_device structure, or NULL if device not found.

void **edac_device_handle_ue**(struct edac_device_ctl_info * *edac_dev*, int *inst_nr*, int *block_nr*, const char * *msg*)

Parameters

struct edac_device_ctl_info * edac_dev pointer to struct edac_device_ctl_info

int inst_nr number of the instance where the UE error happened

int block_nr number of the block where the UE error happened

const char * msg message to be printed

Description

perform a common output and handling of an 'edac_dev' UE event

void **edac_device_handle_ce**(struct edac_device_ctl_info * *edac_dev*, int *inst_nr*, int *block_nr*, const char * *msg*)

Parameters

struct edac_device_ctl_info * edac_dev pointer to struct edac_device_ctl_info

int inst_nr number of the instance where the CE error happened

int block_nr number of the block where the CE error happened

const char * msg message to be printed

Description

perform a common output and handling of an 'edac_dev' CE event

int **edac_device_alloc_index**(void)

Parameters

void no arguments

Return

allocated index number

CHAPTER

TWENTY

SCSI INTERFACES GUIDE

Author James Bottomley

Author Rob Landley

Introduction

Protocol vs bus

Once upon a time, the Small Computer Systems Interface defined both a parallel I/O bus and a data protocol to connect a wide variety of peripherals (disk drives, tape drives, modems, printers, scanners, optical drives, test equipment, and medical devices) to a host computer.

Although the old parallel (fast/wide/ultra) SCSI bus has largely fallen out of use, the SCSI command set is more widely used than ever to communicate with devices over a number of different busses.

The SCSI protocol is a big-endian peer-to-peer packet based protocol. SCSI commands are 6, 10, 12, or 16 bytes long, often followed by an associated data payload.

SCSI commands can be transported over just about any kind of bus, and are the default protocol for storage devices attached to USB, SATA, SAS, Fibre Channel, FireWire, and ATAPI devices. SCSI packets are also commonly exchanged over Infiniband, I2O, TCP/IP (iSCSI), even Parallel ports.

Design of the Linux SCSI subsystem

The SCSI subsystem uses a three layer design, with upper, mid, and low layers. Every operation involving the SCSI subsystem (such as reading a sector from a disk) uses one driver at each of the 3 levels: one upper layer driver, one lower layer driver, and the SCSI midlayer.

The SCSI upper layer provides the interface between userspace and the kernel, in the form of block and char device nodes for I/O and ioctl(). The SCSI lower layer contains drivers for specific hardware devices.

In between is the SCSI mid-layer, analogous to a network routing layer such as the IPv4 stack. The SCSI mid-layer routes a packet based data protocol between the upper layer's /dev nodes and the corresponding devices in the lower layer. It manages command queues, provides error handling and power management functions, and responds to ioctl() requests.

SCSI upper layer

The upper layer supports the user-kernel interface by providing device nodes.

sd (SCSI Disk)

sd (sd_mod.o)

sr (SCSI CD-ROM)

sr (sr_mod.o)

st (SCSI Tape)

st (st.o)

sg (SCSI Generic)

sg (sg.o)

ch (SCSI Media Changer)

ch (ch.c)

SCSI mid layer

SCSI midlayer implementation

include/scsi/scsi_device.h

struct **scsi_vpd**
SCSI Vital Product Data

Definition

```
struct scsi_vpd {
  struct rcu_head rcu;
  int len;
  unsigned char   data[];
};
```

Members

rcu For kfree_rcu().

len Length in bytes of **data**.

data VPD data as defined in various T10 SCSI standard documents.

shost_for_each_device(*sdev*, *shost*)
iterate over all devices of a host

Parameters

sdev the struct scsi_device to use as a cursor

shost the struct scsi_host to iterate over

Description

Iterator that returns each device attached to **shost**. This loop takes a reference on each device and releases it at the end. If you break out of the loop, you must call scsi_device_put(sdev).

__shost_for_each_device(*sdev*, *shost*)
iterate over all devices of a host (UNLOCKED)

Parameters

sdev the struct scsi_device to use as a cursor

shost the `struct scsi_host` to iterate over

Description

Iterator that returns each device attached to **shost**. It does _not_ take a reference on the scsi_device, so the whole loop must be protected by shost->host_lock.

Note

The only reason to use this is because you need to access the device list in interrupt context. Otherwise you really want to use shost_for_each_device instead.

int **scsi_device_supports_vpd**(struct scsi_device * *sdev*)
test if a device supports VPD pages

Parameters

struct scsi_device * sdev the `struct scsi_device` to test

Description

If the 'try_vpd_pages' flag is set it takes precedence. Otherwise we will assume VPD pages are supported if the SCSI level is at least SPC-3 and 'skip_vpd_pages' is not set.

drivers/scsi/scsi.c

Main file for the SCSI midlayer.

void **scsi_cmd_get_serial**(struct Scsi_Host * *host*, struct scsi_cmnd * *cmd*)
Assign a serial number to a command

Parameters

struct Scsi_Host * host the scsi host

struct scsi_cmnd * cmd command to assign serial number to

Description

a serial number identifies a request for error recovery and debugging purposes. Protected by the Host_Lock of host.

int **scsi_change_queue_depth**(struct scsi_device * *sdev*, int *depth*)
change a device's queue depth

Parameters

struct scsi_device * sdev SCSI Device in question

int depth number of commands allowed to be queued to the driver

Description

Sets the device queue depth and returns the new value.

int **scsi_track_queue_full**(struct scsi_device * *sdev*, int *depth*)
track QUEUE_FULL events to adjust queue depth

Parameters

struct scsi_device * sdev SCSI Device in question

int depth Current number of outstanding SCSI commands on this device, not counting the one returned as QUEUE_FULL.

Description

This function will track successive QUEUE_FULL events on a specific SCSI device to determine if and when there is a need to adjust the queue depth on the device.

Return

0 - No change needed, >0 - Adjust queue depth to this new depth,

-1 - Drop back to untagged operation using host->cmd_per_lun as the untagged command depth

Lock Status: None held on entry

Notes

Low level drivers may call this at any time and we will do "The Right Thing." We are interrupt context safe.

int **`scsi_get_vpd_page`**(struct scsi_device * *sdev*, u8 *page*, unsigned char * *buf*, int *buf_len*)
Get Vital Product Data from a SCSI device

Parameters

`struct scsi_device * sdev` The device to ask

`u8 page` Which Vital Product Data to return

`unsigned char * buf` where to store the VPD

`int buf_len` number of bytes in the VPD buffer area

Description

SCSI devices may optionally supply Vital Product Data. Each 'page' of VPD is defined in the appropriate SCSI document (eg SPC, SBC). If the device supports this VPD page, this routine returns a pointer to a buffer containing the data from that page. The caller is responsible for calling `kfree()` on this pointer when it is no longer needed. If we cannot retrieve the VPD page this routine returns NULL.

int **`scsi_report_opcode`**(struct scsi_device * *sdev*, unsigned char * *buffer*, unsigned int *len*, unsigned char *opcode*)
Find out if a given command opcode is supported

Parameters

`struct scsi_device * sdev` scsi device to query

`unsigned char * buffer` scratch buffer (must be at least 20 bytes long)

`unsigned int len` length of buffer

`unsigned char opcode` opcode for command to look up

Description

Uses the REPORT SUPPORTED OPERATION CODES to look up the given opcode. Returns -EINVAL if RSOC fails, 0 if the command opcode is unsupported and 1 if the device claims to support the command.

int **`scsi_device_get`**(struct scsi_device * *sdev*)
get an additional reference to a scsi_device

Parameters

`struct scsi_device * sdev` device to get a reference to

Description

Gets a reference to the scsi_device and increments the use count of the underlying LLDD module. You must hold host_lock of the parent Scsi_Host or already have a reference when calling this.

This will fail if a device is deleted or cancelled, or when the LLD module is in the process of being unloaded.

void **`scsi_device_put`**(struct scsi_device * *sdev*)
release a reference to a scsi_device

Parameters

`struct scsi_device * sdev` device to release a reference on.

Description

Release a reference to the scsi_device and decrements the use count of the underlying LLDD module. The device is freed once the last user vanishes.

void **starget_for_each_device**(struct scsi_target * *starget*, void * *data*, void (*fn) (struct scsi_device *, void *)
helper to walk all devices of a target

Parameters

struct scsi_target * starget target whose devices we want to iterate over.

void * data Opaque passed to each function call.

void (*)(struct scsi_device *, void *) fn Function to call on each device

Description

This traverses over each device of **starget**. The devices have a reference that must be released by scsi_host_put when breaking out of the loop.

void **__starget_for_each_device**(struct scsi_target * *starget*, void * *data*, void (*fn) (struct scsi_device *, void *)
helper to walk all devices of a target (UNLOCKED)

Parameters

struct scsi_target * starget target whose devices we want to iterate over.

void * data parameter for callback **fn()**

void (*)(struct scsi_device *, void *) fn callback function that is invoked for each device

Description

This traverses over each device of **starget**. It does _not_ take a reference on the scsi_device, so the whole loop must be protected by shost->host_lock.

Note

The only reason why drivers would want to use this is because they need to access the device list in irq context. Otherwise you really want to use starget_for_each_device instead.

struct scsi_device * **__scsi_device_lookup_by_target**(struct scsi_target * *starget*, u64 *lun*)
find a device given the target (UNLOCKED)

Parameters

struct scsi_target * starget SCSI target pointer

u64 lun SCSI Logical Unit Number

Description

Looks up the scsi_device with the specified **lun** for a given **starget**. The returned scsi_device does not have an additional reference. You must hold the host's host_lock over this call and any access to the returned scsi_device. A scsi_device in state SDEV_DEL is skipped.

Note

The only reason why drivers should use this is because they need to access the device list in irq context. Otherwise you really want to use scsi_device_lookup_by_target instead.

struct scsi_device * **scsi_device_lookup_by_target**(struct scsi_target * *starget*, u64 *lun*)
find a device given the target

Parameters

struct scsi_target * starget SCSI target pointer

u64 lun SCSI Logical Unit Number

Description

Looks up the scsi_device with the specified **lun** for a given **starget**. The returned scsi_device has an additional reference that needs to be released with scsi_device_put once you're done with it.

struct scsi_device * **`__scsi_device_lookup`**(struct Scsi_Host * *shost*, uint *channel*, uint *id*, u64 *lun*)
find a device given the host (UNLOCKED)

Parameters

`struct Scsi_Host * shost` SCSI host pointer

`uint channel` SCSI channel (zero if only one channel)

`uint id` SCSI target number (physical unit number)

`u64 lun` SCSI Logical Unit Number

Description

Looks up the scsi_device with the specified **channel**, **id**, **lun** for a given host. The returned scsi_device does not have an additional reference. You must hold the host's host_lock over this call and any access to the returned scsi_device.

Note

The only reason why drivers would want to use this is because they need to access the device list in irq context. Otherwise you really want to use scsi_device_lookup instead.

struct scsi_device * **`scsi_device_lookup`**(struct Scsi_Host * *shost*, uint *channel*, uint *id*, u64 *lun*)
find a device given the host

Parameters

`struct Scsi_Host * shost` SCSI host pointer

`uint channel` SCSI channel (zero if only one channel)

`uint id` SCSI target number (physical unit number)

`u64 lun` SCSI Logical Unit Number

Description

Looks up the scsi_device with the specified **channel**, **id**, **lun** for a given host. The returned scsi_device has an additional reference that needs to be released with scsi_device_put once you're done with it.

drivers/scsi/scsicam.c

SCSI Common Access Method support functions, for use with HDIO_GETGEO, etc.

unsigned char * **`scsi_bios_ptable`**(struct block_device * *dev*)
Read PC partition table out of first sector of device.

Parameters

`struct block_device * dev` from this device

Description

Reads the first sector from the device and returns `0x42` bytes starting at offset `0x1be`.

Return

partition table in kmalloc(GFP_KERNEL) memory, or NULL on error.

int **`scsicam_bios_param`**(struct block_device * *bdev*, sector_t *capacity*, int * *ip*)
Determine geometry of a disk in cylinders/heads/sectors.

Parameters

`struct block_device * bdev` which device

`sector_t capacity` size of the disk in sectors

`int * ip` return value: ip[0]=heads, ip[1]=sectors, ip[2]=cylinders

Description

determine the BIOS mapping/geometry used for a drive in a SCSI-CAM system, storing the results in ip as required by the HDIO_GETGEO `ioctl()`.

Return

-1 on failure, 0 on success.

int **`scsi_partsize`**(unsigned char * *buf*, unsigned long *capacity*, unsigned int * *cyls*, unsigned int * *hds*, unsigned int * *secs*)
Parse cylinders/heads/sectors from PC partition table

Parameters

`unsigned char * buf` partition table, see *`scsi_bios_ptable()`*

`unsigned long capacity` size of the disk in sectors

`unsigned int * cyls` put cylinders here

`unsigned int * hds` put heads here

`unsigned int * secs` put sectors here

Description

Determine the BIOS mapping/geometry used to create the partition table, storing the results in **cyls**, **hds**, and **secs**

Return

-1 on failure, 0 on success.

drivers/scsi/scsi_error.c

Common SCSI error/timeout handling routines.

void **`scsi_schedule_eh`**(struct Scsi_Host * *shost*)
schedule EH for SCSI host

Parameters

`struct Scsi_Host * shost` SCSI host to invoke error handling on.

Description

Schedule SCSI EH without scmd.

int **`scsi_block_when_processing_errors`**(struct scsi_device * *sdev*)
Prevent cmds from being queued.

Parameters

`struct scsi_device * sdev` Device on which we are performing recovery.

Description

We block until the host is out of error recovery, and then check to see whether the host or the device is offline.

Return value: 0 when dev was taken offline by error recovery. 1 OK to proceed.

int **`scsi_check_sense`**(struct scsi_cmnd * *scmd*)
Examine scsi cmd sense

Parameters

`struct scsi_cmnd * scmd` Cmd to have sense checked.

Description

Return value: SUCCESS or FAILED or NEEDS_RETRY or ADD_TO_MLQUEUE

Notes

When a deferred error is detected the current command has not been executed and needs retrying.

void **`scsi_eh_prep_cmnd`**(struct scsi_cmnd * *scmd*, struct scsi_eh_save * *ses*, unsigned char * *cmnd*, int *cmnd_size*, unsigned *sense_bytes*)
Save a scsi command info as part of error recovery

Parameters

`struct scsi_cmnd * scmd` SCSI command structure to hijack

`struct scsi_eh_save * ses` structure to save restore information

`unsigned char * cmnd` CDB to send. Can be NULL if no new cmnd is needed

`int cmnd_size` size in bytes of **cmnd** (must be <= BLK_MAX_CDB)

`unsigned sense_bytes` size of sense data to copy. or 0 (if != 0 **cmnd** is ignored)

Description

This function is used to save a scsi command information before re-execution as part of the error recovery process. If **sense_bytes** is 0 the command sent must be one that does not transfer any data. If **sense_bytes** != 0 **cmnd** is ignored and this functions sets up a REQUEST_SENSE command and cmnd buffers to read **sense_bytes** into **scmd**->sense_buffer.

void **`scsi_eh_restore_cmnd`**(struct scsi_cmnd * *scmd*, struct scsi_eh_save * *ses*)
Restore a scsi command info as part of error recovery

Parameters

`struct scsi_cmnd * scmd` SCSI command structure to restore

`struct scsi_eh_save * ses` saved information from a coresponding call to scsi_eh_prep_cmnd

Description

Undo any damage done by above *`scsi_eh_prep_cmnd()`*.

void **`scsi_eh_finish_cmd`**(struct scsi_cmnd * *scmd*, struct list_head * *done_q*)
Handle a cmd that eh is finished with.

Parameters

`struct scsi_cmnd * scmd` Original SCSI cmd that eh has finished.

`struct list_head * done_q` Queue for processed commands.

Notes

We don't want to use the normal command completion while we are are still handling errors - it may cause other commands to be queued, and that would disturb what we are doing. Thus we really want to keep a list of pending commands for final completion, and once we are ready to leave error handling we handle completion for real.

int **`scsi_eh_get_sense`**(struct list_head * *work_q*, struct list_head * *done_q*)
Get device sense data.

Parameters

`struct list_head * work_q` Queue of commands to process.

`struct list_head * done_q` Queue of processed commands.

Description

See if we need to request sense information. if so, then get it now, so we have a better idea of what to do.

Notes

This has the unfortunate side effect that if a shost adapter does not automatically request sense information, we end up shutting it down before we request it.

All drivers should request sense information internally these days, so for now all I have to say is tough noogies if you end up in here.

XXX: Long term this code should go away, but that needs an audit of all LLDDs first.

void **scsi_eh_ready_devs**(struct Scsi_Host * *shost*, struct list_head * *work_q*, struct list_head * *done_q*)
check device ready state and recover if not.

Parameters

struct Scsi_Host * shost host to be recovered.

struct list_head * work_q `list_head` for pending commands.

struct list_head * done_q `list_head` for processed commands.

void **scsi_eh_flush_done_q**(struct list_head * *done_q*)
finish processed commands or retry them.

Parameters

struct list_head * done_q list_head of processed commands.

int **scsi_ioctl_reset**(struct scsi_device * *dev*, int __user * *arg*)

Parameters

struct scsi_device * dev scsi_device to operate on

int __user * arg reset type (see sg.h)

bool **scsi_get_sense_info_fld**(const u8 * *sense_buffer*, int *sb_len*, u64 * *info_out*)
get information field from sense data (either fixed or descriptor format)

Parameters

const u8 * sense_buffer byte array of sense data

int sb_len number of valid bytes in sense_buffer

u64 * info_out pointer to 64 integer where 8 or 4 byte information field will be placed if found.

Description

Return value: true if information field found, false if not found.

drivers/scsi/scsi_devinfo.c

Manage scsi_dev_info_list, which tracks blacklisted and whitelisted devices.

int **scsi_dev_info_list_add**(int *compatible*, char * *vendor*, char * *model*, char * *strflags*, blist_flags_t *flags*)
add one dev_info list entry.

Parameters

int compatible if true, null terminate short strings. Otherwise space pad.

char * vendor vendor string

char * model model (product) string

char * strflags integer string

blist_flags_t flags if strflags NULL, use this flag value

Description

Create and add one dev_info entry for **vendor**, **model**, **strflags** or **flag**. If **compatible**, add to the tail of the list, do not space pad, and set devinfo->compatible. The scsi_static_device_list entries are added with **compatible** 1 and **clfags** NULL.

Return

0 OK, -error on failure.

struct scsi_dev_info_list * **scsi_dev_info_list_find**(const char * *vendor*, const char * *model*, enum scsi_devinfo_key *key*)
find a matching dev_info list entry.

Parameters

const char * vendor full vendor string

const char * model full model (product) string

enum scsi_devinfo_key key specify list to use

Description

Finds the first dev_info entry matching **vendor**, **model** in list specified by **key**.

Return

pointer to matching entry, or ERR_PTR on failure.

int **scsi_dev_info_list_add_str**(char * *dev_list*)
parse dev_list and add to the scsi_dev_info_list.

Parameters

char * dev_list string of device flags to add

Description

Parse dev_list, and add entries to the scsi_dev_info_list. dev_list is of the form "vendor:product:flag,vendor:product:flag". dev_list is modified via strsep. Can be called for command line addition, for proc or mabye a sysfs interface.

Return

0 if OK, -error on failure.

blist_flags_t **scsi_get_device_flags**(struct scsi_device * *sdev*, const unsigned char * *vendor*, const unsigned char * *model*)
get device specific flags from the dynamic device list.

Parameters

struct scsi_device * sdev scsi_device to get flags for

const unsigned char * vendor vendor name

const unsigned char * model model name

Description

Search the global scsi_dev_info_list (specified by list zero) for an entry matching **vendor** and **model**, if found, return the matching flags value, else return the host or global default settings. Called during scan time.

void **scsi_exit_devinfo**(void)
remove /proc/scsi/device_info & the scsi_dev_info_list

Parameters

void no arguments

int **scsi_init_devinfo**(void)
set up the dynamic device list.

Parameters

void no arguments

Description

Add command line entries from scsi_dev_flags, then add scsi_static_device_list entries to the scsi device info list.

drivers/scsi/scsi_ioctl.c

Handle ioctl() calls for SCSI devices.

int **scsi_ioctl**(struct scsi_device * *sdev*, int *cmd*, void __user * *arg*)
Dispatch ioctl to scsi device

Parameters

struct scsi_device * sdev scsi device receiving ioctl

int cmd which ioctl is it

void __user * arg data associated with ioctl

Description

The *scsi_ioctl()* function differs from most ioctls in that it does not take a major/minor number as the dev field. Rather, it takes a pointer to a `struct scsi_device`.

drivers/scsi/scsi_lib.c

SCSI queuing library.

int **scsi_execute**(struct scsi_device * *sdev*, const unsigned char * *cmd*, int *data_direction*, void * *buffer*, unsigned *bufflen*, unsigned char * *sense*, struct scsi_sense_hdr * *sshdr*, int *timeout*, int *retries*, u64 *flags*, req_flags_t *rq_flags*, int * *resid*)
insert request and wait for the result

Parameters

struct scsi_device * sdev scsi device

const unsigned char * cmd scsi command

int data_direction data direction

void * buffer data buffer

unsigned bufflen len of buffer

unsigned char * sense optional sense buffer

struct scsi_sense_hdr * sshdr optional decoded sense header

int timeout request timeout in seconds

int retries number of times to retry request

u64 flags flags for ->cmd_flags

req_flags_t rq_flags flags for ->rq_flags

int * resid optional residual length

Description

Returns the scsi_cmnd result field if a command was executed, or a negative Linux error code if we didn't get that far.

struct scsi_device * **scsi_device_from_queue**(struct request_queue * *q*)
return sdev associated with a request_queue

Parameters

struct request_queue * q The request queue to return the sdev from

Description

Return the sdev associated with a request queue or NULL if the request_queue does not reference a SCSI device.

int **scsi_mode_select**(struct scsi_device * *sdev*, int *pf*, int *sp*, int *modepage*, unsigned char * *buffer*, int *len*, int *timeout*, int *retries*, struct scsi_mode_data * *data*, struct scsi_sense_hdr * *sshdr*)
issue a mode select

Parameters

struct scsi_device * sdev SCSI device to be queried

int pf Page format bit (1 == standard, 0 == vendor specific)

int sp Save page bit (0 == don't save, 1 == save)

int modepage mode page being requested

unsigned char * buffer request buffer (may not be smaller than eight bytes)

int len length of request buffer.

int timeout command timeout

int retries number of retries before failing

struct scsi_mode_data * data returns a structure abstracting the mode header data

struct scsi_sense_hdr * sshdr place to put sense data (or NULL if no sense to be collected). must be SCSI_SENSE_BUFFERSIZE big.

Description

Returns zero if successful; negative error number or scsi status on error

int **scsi_mode_sense**(struct scsi_device * *sdev*, int *dbd*, int *modepage*, unsigned char * *buffer*, int *len*, int *timeout*, int *retries*, struct scsi_mode_data * *data*, struct scsi_sense_hdr * *sshdr*)
issue a mode sense, falling back from 10 to six bytes if necessary.

Parameters

struct scsi_device * sdev SCSI device to be queried

int dbd set if mode sense will allow block descriptors to be returned

int modepage mode page being requested

unsigned char * buffer request buffer (may not be smaller than eight bytes)

int len length of request buffer.

int timeout command timeout

int retries number of retries before failing

struct scsi_mode_data * data returns a structure abstracting the mode header data

struct scsi_sense_hdr * sshdr place to put sense data (or NULL if no sense to be collected). must be SCSI_SENSE_BUFFERSIZE big.

Description

Returns zero if unsuccessful, or the header offset (either 4 or 8 depending on whether a six or ten byte command was issued) if successful.

int **scsi_test_unit_ready**(struct scsi_device * *sdev*, int *timeout*, int *retries*, struct scsi_sense_hdr * *sshdr*)
test if unit is ready

Parameters

struct scsi_device * sdev scsi device to change the state of.

int timeout command timeout

int retries number of retries before failing

struct scsi_sense_hdr * sshdr outpout pointer for decoded sense information.

Description

Returns zero if unsuccessful or an error if TUR failed. For removable media, UNIT_ATTENTION sets ->changed flag.

int **scsi_device_set_state**(struct scsi_device * *sdev*, enum scsi_device_state *state*)
Take the given device through the device state model.

Parameters

struct scsi_device * sdev scsi device to change the state of.

enum scsi_device_state state state to change to.

Description

Returns zero if successful or an error if the requested transition is illegal.

void **sdev_evt_send**(struct scsi_device * *sdev*, struct scsi_event * *evt*)
send asserted event to uevent thread

Parameters

struct scsi_device * sdev scsi_device event occurred on

struct scsi_event * evt event to send

Description

Assert scsi device event asynchronously.

struct scsi_event * **sdev_evt_alloc**(enum scsi_device_event *evt_type*, gfp_t *gfpflags*)
allocate a new scsi event

Parameters

enum scsi_device_event evt_type type of event to allocate

gfp_t gfpflags GFP flags for allocation

Description

Allocates and returns a new scsi_event.

void **sdev_evt_send_simple**(struct scsi_device * *sdev*, enum scsi_device_event *evt_type*, gfp_t *gfpflags*)
send asserted event to uevent thread

Parameters

struct scsi_device * sdev scsi_device event occurred on

enum scsi_device_event evt_type type of event to send

gfp_t gfpflags GFP flags for allocation

Description

Assert scsi device event asynchronously, given an event type.

int **scsi_device_quiesce**(struct scsi_device * *sdev*)
Block user issued commands.

Parameters

struct scsi_device * sdev scsi device to quiesce.

Description

This works by trying to transition to the SDEV_QUIESCE state (which must be a legal transition). When the device is in this state, only special requests will be accepted, all others will be deferred. Since special requests may also be requeued requests, a successful return doesn't guarantee the device will be totally quiescent.

Must be called with user context, may sleep.

Returns zero if unsuccessful or an error if not.

void **scsi_device_resume**(struct scsi_device * *sdev*)
Restart user issued commands to a quiesced device.

Parameters

struct scsi_device * sdev scsi device to resume.

Description

Moves the device from quiesced back to running and restarts the queues.

Must be called with user context, may sleep.

int **scsi_internal_device_block_nowait**(struct scsi_device * *sdev*)
try to transition to the SDEV_BLOCK state

Parameters

struct scsi_device * sdev device to block

Description

Pause SCSI command processing on the specified device. Does not sleep.

Returns zero if successful or a negative error code upon failure.

Notes

This routine transitions the device to the SDEV_BLOCK state (which must be a legal transition). When the device is in this state, command processing is paused until the device leaves the SDEV_BLOCK state. See also *scsi_internal_device_unblock_nowait()*.

int **scsi_internal_device_unblock_nowait**(struct scsi_device * *sdev*, enum scsi_device_state *new_state*)
resume a device after a block request

Parameters

struct scsi_device * sdev device to resume

enum scsi_device_state new_state state to set the device to after unblocking

Description

Restart the device queue for a previously suspended SCSI device. Does not sleep.

Returns zero if successful or a negative error code upon failure.

Notes

This routine transitions the device to the SDEV_RUNNING state or to one of the offline states (which must be a legal transition) allowing the midlayer to goose the queue for this device.

void * **scsi_kmap_atomic_sg**(struct scatterlist * *sgl*, int *sg_count*, size_t * *offset*, size_t * *len*)
find and atomically map an sg-elemnt

Parameters

struct scatterlist * sgl scatter-gather list

int sg_count number of segments in sg

size_t * offset offset in bytes into sg, on return offset into the mapped area

size_t * len bytes to map, on return number of bytes mapped

Description

Returns virtual address of the start of the mapped page

void **scsi_kunmap_atomic_sg**(void * *virt*)
atomically unmap a virtual address, previously mapped with scsi_kmap_atomic_sg

Parameters

void * virt virtual address to be unmapped

int **scsi_vpd_lun_id**(struct scsi_device * *sdev*, char * *id*, size_t *id_len*)
return a unique device identification

Parameters

struct scsi_device * sdev SCSI device

char * id buffer for the identification

size_t id_len length of the buffer

Description

Copies a unique device identification into **id** based on the information in the VPD page 0x83 of the device. The string will be formatted as a SCSI name string.

Returns the length of the identification or error on failure. If the identifier is longer than the supplied buffer the actual identifier length is returned and the buffer is not zero-padded.

drivers/scsi/scsi_lib_dma.c

SCSI library functions depending on DMA (map and unmap scatter-gather lists).

int **scsi_dma_map**(struct scsi_cmnd * *cmd*)
perform DMA mapping against command's sg lists

Parameters

struct scsi_cmnd * cmd scsi command

Description

Returns the number of sg lists actually used, zero if the sg lists is NULL, or -ENOMEM if the mapping failed.

void **scsi_dma_unmap**(struct scsi_cmnd * *cmd*)
unmap command's sg lists mapped by scsi_dma_map

Parameters

struct scsi_cmnd * cmd scsi command

drivers/scsi/scsi_module.c

The file drivers/scsi/scsi_module.c contains legacy support for old-style host templates. It should never be used by any new driver.

drivers/scsi/scsi_proc.c

The functions in this file provide an interface between the PROC file system and the SCSI device drivers It is mainly used for debugging, statistics and to pass information directly to the lowlevel driver. I.E. plumbing to manage /proc/scsi/*

void **scsi_proc_hostdir_add**(struct scsi_host_template * *sht*)
Create directory in /proc for a scsi host

Parameters

struct scsi_host_template * sht owner of this directory

Description

Sets sht->proc_dir to the new directory.

void **scsi_proc_hostdir_rm**(struct scsi_host_template * *sht*)
remove directory in /proc for a scsi host

Parameters

struct scsi_host_template * sht owner of directory

void **scsi_proc_host_add**(struct Scsi_Host * *shost*)
Add entry for this host to appropriate /proc dir

Parameters

struct Scsi_Host * shost host to add

void **scsi_proc_host_rm**(struct Scsi_Host * *shost*)
remove this host's entry from /proc

Parameters

struct Scsi_Host * shost which host

int **proc_print_scsidevice**(struct *device* * *dev*, void * *data*)
return data about this host

Parameters

struct device * dev A scsi device

void * data `struct seq_file` to output to.

Description

prints Host, Channel, Id, Lun, Vendor, Model, Rev, Type, and revision.

int **scsi_add_single_device**(uint *host*, uint *channel*, uint *id*, uint *lun*)
Respond to user request to probe for/add device

Parameters

uint host user-supplied decimal integer

uint channel user-supplied decimal integer

uint id user-supplied decimal integer

uint lun user-supplied decimal integer

Description

called by writing "scsi add-single-device" to /proc/scsi/scsi.

does *scsi_host_lookup()* and either `user_scan()` if that transport type supports it, or else `scsi_scan_host_selected()`

Note

this seems to be aimed exclusively at SCSI parallel busses.

int **scsi_remove_single_device**(uint *host*, uint *channel*, uint *id*, uint *lun*)
Respond to user request to remove a device

Parameters

uint host user-supplied decimal integer

uint channel user-supplied decimal integer

uint id user-supplied decimal integer

uint lun user-supplied decimal integer

Description

called by writing “scsi remove-single-device” to /proc/scsi/scsi. Does a *scsi_device_lookup()* and *scsi_remove_device()*

ssize_t **proc_scsi_write**(struct file * *file*, const char __user * *buf*, size_t *length*, loff_t * *ppos*)
handle writes to /proc/scsi/scsi

Parameters

struct file * file not used

const char __user * buf buffer to write

size_t length length of buf, at most PAGE_SIZE

loff_t * ppos not used

Description

this provides a legacy mechanism to add or remove devices by Host, Channel, ID, and Lun. To use, “echo ‘scsi add-single-device 0 1 2 3’ > /proc/scsi/scsi” or “echo ‘scsi remove-single-device 0 1 2 3’ > /proc/scsi/scsi” with “0 1 2 3” replaced by the Host, Channel, Id, and Lun.

Note

this seems to be aimed at parallel SCSI. Most modern busses (USB, SATA, Firewire, Fibre Channel, etc) dynamically assign these values to provide a unique identifier and nothing more.

int **proc_scsi_open**(struct inode * *inode*, struct file * *file*)
glue function

Parameters

struct inode * inode not used

struct file * file passed to `single_open()`

Description

Associates proc_scsi_show with this file

int **scsi_init_procfs**(void)
create scsi and scsi/scsi in procfs

Parameters

void no arguments

void **scsi_exit_procfs**(void)
Remove scsi/scsi and scsi from procfs

Parameters

void no arguments

drivers/scsi/scsi_netlink.c

Infrastructure to provide async events from transports to userspace via netlink, using a single NETLINK_SCSITRANSPORT protocol for all transports. See the original patch submission for more details.

void **scsi_nl_rcv_msg**(struct sk_buff * *skb*)
Receive message handler.

Parameters

struct sk_buff * skb socket receive buffer

Description

Extracts message from a receive buffer. Validates message header and calls appropriate transport message handler

void **scsi_netlink_init**(void)
Called by SCSI subsystem to initialize the SCSI transport netlink interface

Parameters

void no arguments

void **scsi_netlink_exit**(void)
Called by SCSI subsystem to disable the SCSI transport netlink interface

Parameters

void no arguments

drivers/scsi/scsi_scan.c

Scan a host to determine which (if any) devices are attached. The general scanning/probing algorithm is as follows, exceptions are made to it depending on device specific flags, compilation options, and global variable (boot or module load time) settings. A specific LUN is scanned via an INQUIRY command; if the LUN has a device attached, a scsi_device is allocated and setup for it. For every id of every channel on the given host, start by scanning LUN 0. Skip hosts that don't respond at all to a scan of LUN 0. Otherwise, if LUN 0 has a device attached, allocate and setup a scsi_device for it. If target is SCSI-3 or up, issue a REPORT LUN, and scan all of the LUNs returned by the REPORT LUN; else, sequentially scan LUNs up until some maximum is reached, or a LUN is seen that cannot have a device attached to it.

int **scsi_complete_async_scans**(void)
Wait for asynchronous scans to complete

Parameters

void no arguments

Description

When this function returns, any host which started scanning before this function was called will have finished its scan. Hosts which started scanning after this function was called may or may not have finished.

void **scsi_unlock_floptical**(struct scsi_device * *sdev*, unsigned char * *result*)
unlock device via a special MODE SENSE command

Parameters

struct scsi_device * sdev scsi device to send command to

unsigned char * result area to store the result of the MODE SENSE

Description

Send a vendor specific MODE SENSE (not a MODE SELECT) command. Called for BLIST_KEY devices.

struct scsi_device * **scsi_alloc_sdev**(struct scsi_target * *starget*, u64 *lun*, void * *hostdata*)
allocate and setup a scsi_Device

Parameters

struct scsi_target * starget which target to allocate a `scsi_device` for

u64 lun which lun

void * hostdata usually NULL and set by ->slave_alloc instead

Description

Allocate, initialize for io, and return a pointer to a scsi_Device. Stores the **shost**, **channel**, **id**, and **lun** in the scsi_Device, and adds scsi_Device to the appropriate list.

Return value: scsi_Device pointer, or NULL on failure.

void **scsi_target_reap_ref_release**(struct kref * *kref*)
remove target from visibility

Parameters

struct kref * kref the reap_ref in the target being released

Description

Called on last put of reap_ref, which is the indication that no device under this target is visible anymore, so render the target invisible in sysfs. Note: we have to be in user context here because the target reaps should be done in places where the scsi device visibility is being removed.

struct scsi_target * **scsi_alloc_target**(struct *device* * *parent*, int *channel*, uint *id*)
allocate a new or find an existing target

Parameters

struct device * parent parent of the target (need not be a scsi host)

int channel target channel number (zero if no channels)

uint id target id number

Description

Return an existing target if one exists, provided it hasn't already gone into STARGET_DEL state, otherwise allocate a new target.

The target is returned with an incremented reference, so the caller is responsible for both reaping and doing a last put

void **scsi_target_reap**(struct scsi_target * *starget*)
check to see if target is in use and destroy if not

Parameters

struct scsi_target * starget target to be checked

Description

This is used after removing a LUN or doing a last put of the target it checks atomically that nothing is using the target and removes it if so.

int **scsi_probe_lun**(struct scsi_device * *sdev*, unsigned char * *inq_result*, int *result_len*, blist_flags_t * *bflags*)
probe a single LUN using a SCSI INQUIRY

Parameters

struct scsi_device * sdev scsi_device to probe

unsigned char * inq_result area to store the INQUIRY result

int result_len len of inq_result

blist_flags_t * bflags store any bflags found here

Description

Probe the lun associated with **req** using a standard SCSI INQUIRY;

If the INQUIRY is successful, zero is returned and the INQUIRY data is in **inq_result**; the scsi_level and INQUIRY length are copied to the scsi_device any flags value is stored in ***bflags**.

int **scsi_add_lun**(struct scsi_device * *sdev*, unsigned char * *inq_result*, blist_flags_t * *bflags*, int *async*)
allocate and fully initialze a scsi_device

Parameters

struct scsi_device * sdev holds information to be stored in the new scsi_device

unsigned char * inq_result holds the result of a previous INQUIRY to the LUN

blist_flags_t * bflags black/white list flag

int async 1 if this device is being scanned asynchronously

Description

Initialize the scsi_device **sdev**. Optionally set fields based on values in ***bflags**.

Return

SCSI_SCAN_NO_RESPONSE: could not allocate or setup a scsi_device SCSI_SCAN_LUN_PRESENT: a new scsi_device was allocated and initialized

unsigned char * **scsi_inq_str**(unsigned char * *buf*, unsigned char * *inq*, unsigned *first*, unsigned *end*)
print INQUIRY data from min to max index, strip trailing whitespace

Parameters

unsigned char * buf Output buffer with at least end-first+1 bytes of space

unsigned char * inq Inquiry buffer (input)

unsigned first Offset of string into inq

unsigned end Index after last character in inq

int **scsi_probe_and_add_lun**(struct scsi_target * *starget*, u64 *lun*, blist_flags_t * *bflagsp*, struct scsi_device ** *sdevp*, enum scsi_scan_mode *rescan*, void * *hostdata*)
probe a LUN, if a LUN is found add it

Parameters

struct scsi_target * starget pointer to target device structure

u64 lun LUN of target device

blist_flags_t * bflagsp store bflags here if not NULL

struct scsi_device ** sdevp probe the LUN corresponding to this scsi_device

enum scsi_scan_mode rescan if not equal to SCSI_SCAN_INITIAL skip some code only needed on first scan

void * hostdata passed to *scsi_alloc_sdev()*

Description

Call scsi_probe_lun, if a LUN with an attached device is found, allocate and set it up by calling scsi_add_lun.

Return

- SCSI_SCAN_NO_RESPONSE: could not allocate or setup a scsi_device

- **SCSI_SCAN_TARGET_PRESENT: target responded, but no device is** attached at the LUN
- SCSI_SCAN_LUN_PRESENT: a new scsi_device was allocated and initialized

void **scsi_sequential_lun_scan**(struct scsi_target * *starget*, blist_flags_t *bflags*, int *scsi_level*, enum scsi_scan_mode *rescan*)
sequentially scan a SCSI target

Parameters

struct scsi_target * starget pointer to target structure to scan

blist_flags_t bflags black/white list flag for LUN 0

int scsi_level Which version of the standard does this device adhere to

enum scsi_scan_mode rescan passed to scsi_probe_add_lun()

Description

Generally, scan from LUN 1 (LUN 0 is assumed to already have been scanned) to some maximum lun until a LUN is found with no device attached. Use the bflags to figure out any oddities.

Modifies sdevscan->lun.

int **scsi_report_lun_scan**(struct scsi_target * *starget*, blist_flags_t *bflags*, enum scsi_scan_mode *rescan*)
Scan using SCSI REPORT LUN results

Parameters

struct scsi_target * starget which target

blist_flags_t bflags Zero or a mix of BLIST_NOLUN, BLIST_REPORTLUN2, or BLIST_NOREPORTLUN

enum scsi_scan_mode rescan nonzero if we can skip code only needed on first scan

Description

Fast scanning for modern (SCSI-3) devices by sending a REPORT LUN command. Scan the resulting list of LUNs by calling scsi_probe_and_add_lun.

If BLINK_REPORTLUN2 is set, scan a target that supports more than 8 LUNs even if it's older than SCSI-3. If BLIST_NOREPORTLUN is set, return 1 always. If BLIST_NOLUN is set, return 0 always. If starget->no_report_luns is set, return 1 always.

Return

0: scan completed (or no memory, so further scanning is futile) 1: could not scan with REPORT LUN

struct async_scan_data * **scsi_prep_async_scan**(struct Scsi_Host * *shost*)
prepare for an async scan

Parameters

struct Scsi_Host * shost the host which will be scanned

Return

a cookie to be passed to *scsi_finish_async_scan()*

Tells the midlayer this host is going to do an asynchronous scan. It reserves the host's position in the scanning list and ensures that other asynchronous scans started after this one won't affect the ordering of the discovered devices.

void **scsi_finish_async_scan**(struct async_scan_data * *data*)
asynchronous scan has finished

Parameters

struct async_scan_data * data cookie returned from earlier call to *scsi_prep_async_scan()*

Description

All the devices currently attached to this host have been found. This function announces all the devices it has found to the rest of the system.

drivers/scsi/scsi_sysctl.c

Set up the sysctl entry: "/dev/scsi/logging_level" (DEV_SCSI_LOGGING_LEVEL) which sets/returns scsi_logging_level.

drivers/scsi/scsi_sysfs.c

SCSI sysfs interface routines.

void **scsi_remove_device**(struct scsi_device * *sdev*)
unregister a device from the scsi bus

Parameters

struct scsi_device * sdev scsi_device to unregister

void **scsi_remove_target**(struct *device* * *dev*)
try to remove a target and all its devices

Parameters

struct device * dev generic starget or parent of generic stargets to be removed

Note

This is slightly racy. It is possible that if the user requests the addition of another device then the target won't be removed.

drivers/scsi/hosts.c

mid to lowlevel SCSI driver interface

int **scsi_host_set_state**(struct Scsi_Host * *shost*, enum scsi_host_state *state*)
Take the given host through the host state model.

Parameters

struct Scsi_Host * shost scsi host to change the state of.

enum scsi_host_state state state to change to.

Description

Returns zero if unsuccessful or an error if the requested transition is illegal.

void **scsi_remove_host**(struct Scsi_Host * *shost*)
remove a scsi host

Parameters

struct Scsi_Host * shost a pointer to a scsi host to remove

int **scsi_add_host_with_dma**(struct Scsi_Host * *shost*, struct *device* * *dev*, struct *device* * *dma_dev*)
add a scsi host with dma device

Parameters

struct Scsi_Host * shost scsi host pointer to add

struct device * dev a struct device of type scsi class

struct device * dma_dev dma device for the host

Note

You rarely need to worry about this unless you're in a virtualised host environments, so use the simpler `scsi_add_host()` function instead.

Return value: 0 on success / != 0 for error

struct Scsi_Host * **scsi_host_alloc**(struct scsi_host_template * *sht*, int *privsize*)
register a scsi host adapter instance.

Parameters

struct scsi_host_template * sht pointer to scsi host template

int privsize extra bytes to allocate for driver

Note

Allocate a new Scsi_Host and perform basic initialization. The host is not published to the scsi midlayer until scsi_add_host is called.

Return value: Pointer to a new Scsi_Host

struct Scsi_Host * **scsi_host_lookup**(unsigned short *hostnum*)
get a reference to a Scsi_Host by host no

Parameters

unsigned short hostnum host number to locate

Description

Return value: A pointer to located Scsi_Host or NULL.

The caller must do a *scsi_host_put()* to drop the reference that *scsi_host_get()* took. The *put_device()* below dropped the reference from *class_find_device()*.

struct Scsi_Host * **scsi_host_get**(struct Scsi_Host * *shost*)
inc a Scsi_Host ref count

Parameters

struct Scsi_Host * shost Pointer to Scsi_Host to inc.

void **scsi_host_put**(struct Scsi_Host * *shost*)
dec a Scsi_Host ref count

Parameters

struct Scsi_Host * shost Pointer to Scsi_Host to dec.

int **scsi_queue_work**(struct Scsi_Host * *shost*, struct work_struct * *work*)
Queue work to the Scsi_Host workqueue.

Parameters

struct Scsi_Host * shost Pointer to Scsi_Host.

struct work_struct * work Work to queue for execution.

Description

Return value: 1 - work queued for execution 0 - work is already queued -EINVAL - work queue doesn't exist

void **scsi_flush_work**(struct Scsi_Host * *shost*)
Flush a Scsi_Host's workqueue.

Parameters

struct Scsi_Host * shost Pointer to Scsi_Host.

drivers/scsi/scsi_common.c

general support functions

const char * **`scsi_device_type`**(unsigned *type*)
Return 17-char string indicating device type.

Parameters

`unsigned type` type number to look up

u64 **`scsilun_to_int`**(struct scsi_lun * *scsilun*)
convert a scsi_lun to an int

Parameters

`struct scsi_lun * scsilun` struct scsi_lun to be converted.

Description

Convert **scsilun** from a struct scsi_lun to a four-byte host byte-ordered integer, and return the result. The caller must check for truncation before using this function.

Notes

For a description of the LUN format, post SCSI-3 see the SCSI Architecture Model, for SCSI-3 see the SCSI Controller Commands.

Given a struct scsi_lun of: d2 04 0b 03 00 00 00 00, this function returns the integer: 0x0b03d204

This encoding will return a standard integer LUN for LUNs smaller than 256, which typically use a single level LUN structure with addressing method 0.

void **`int_to_scsilun`**(u64 *lun*, struct scsi_lun * *scsilun*)
reverts an int into a scsi_lun

Parameters

`u64 lun` integer to be reverted

`struct scsi_lun * scsilun` struct scsi_lun to be set.

Description

Reverts the functionality of the scsilun_to_int, which packed an 8-byte lun value into an int. This routine unpacks the int back into the lun value.

Notes

Given an integer : 0x0b03d204, this function returns a struct scsi_lun of: d2 04 0b 03 00 00 00 00

bool **`scsi_normalize_sense`**(const u8 * *sense_buffer*, int *sb_len*, struct scsi_sense_hdr * *sshdr*)
normalize main elements from either fixed or descriptor sense data format into a common format.

Parameters

`const u8 * sense_buffer` byte array containing sense data returned by device

`int sb_len` number of valid bytes in sense_buffer

`struct scsi_sense_hdr * sshdr` pointer to instance of structure that common elements are written to.

Notes

The "main elements" from sense data are: response_code, sense_key, asc, ascq and additional_length (only for descriptor format).

Typically this function can be called after a device has responded to a SCSI command with the CHECK_CONDITION status.

Return value: true if valid sense data information found, else false;

const u8 * **scsi_sense_desc_find**(const u8 * *sense_buffer*, int *sb_len*, int *desc_type*)
search for a given descriptor type in descriptor sense data format.

Parameters

const u8 * sense_buffer byte array of descriptor format sense data

int sb_len number of valid bytes in sense_buffer

int desc_type value of descriptor type to find (e.g. 0 -> information)

Notes

only valid when sense data is in descriptor format

Return value: pointer to start of (first) descriptor if found else NULL

void **scsi_build_sense_buffer**(int *desc*, u8 * *buf*, u8 *key*, u8 *asc*, u8 *ascq*)
build sense data in a buffer

Parameters

int desc Sense format (non-zero == descriptor format, 0 == fixed format)

u8 * buf Where to build sense data

u8 key Sense key

u8 asc Additional sense code

u8 ascq Additional sense code qualifier

int **scsi_set_sense_information**(u8 * *buf*, int *buf_len*, u64 *info*)
set the information field in a formatted sense data buffer

Parameters

u8 * buf Where to build sense data

int buf_len buffer length

u64 info 64-bit information value to be set

Description

Return value: 0 on success or -EINVAL for invalid sense buffer length

int **scsi_set_sense_field_pointer**(u8 * *buf*, int *buf_len*, u16 *fp*, u8 *bp*, bool *cd*)
set the field pointer sense key specific information in a formatted sense data buffer

Parameters

u8 * buf Where to build sense data

int buf_len buffer length

u16 fp field pointer to be set

u8 bp bit pointer to be set

bool cd command/data bit

Description

Return value: 0 on success or -EINVAL for invalid sense buffer length

Transport classes

Transport classes are service libraries for drivers in the SCSI lower layer, which expose transport attributes in sysfs.

Fibre Channel transport

The file drivers/scsi/scsi_transport_fc.c defines transport attributes for Fibre Channel.

u32 **fc_get_event_number**(void)
Obtain the next sequential FC event number

Parameters

void no arguments

Notes

We could have inlined this, but it would have required fc_event_seq to be exposed. For now, live with the subroutine call. Atomic used to avoid lock/unlock…

void **fc_host_post_event**(struct Scsi_Host * *shost*, u32 *event_number*, enum fc_host_event_code *event_code*, u32 *event_data*)
called to post an even on an fc_host.

Parameters

struct Scsi_Host * shost host the event occurred on

u32 event_number fc event number obtained from `get_fc_event_number()`

enum fc_host_event_code event_code fc_host event being posted

u32 event_data 32bits of data for the event being posted

Notes

This routine assumes no locks are held on entry.

void **fc_host_post_vendor_event**(struct Scsi_Host * *shost*, u32 *event_number*, u32 *data_len*, char * *data_buf*, u64 *vendor_id*)
called to post a vendor unique event on an fc_host

Parameters

struct Scsi_Host * shost host the event occurred on

u32 event_number fc event number obtained from `get_fc_event_number()`

u32 data_len amount, in bytes, of vendor unique data

char * data_buf pointer to vendor unique data

u64 vendor_id Vendor id

Notes

This routine assumes no locks are held on entry.

enum blk_eh_timer_return **fc_eh_timed_out**(struct scsi_cmnd * *scmd*)
FC Transport I/O timeout intercept handler

Parameters

struct scsi_cmnd * scmd The SCSI command which timed out

Description

This routine protects against error handlers getting invoked while a rport is in a blocked state, typically due to a temporarily loss of connectivity. If the error handlers are allowed to proceed, requests to abort i/o, reset the target, etc will likely fail as there is no way to communicate with the device to perform the requested function. These failures may result in the midlayer taking the device offline, requiring manual intervention to restore operation.

This routine, called whenever an i/o times out, validates the state of the underlying rport. If the rport is blocked, it returns EH_RESET_TIMER, which will continue to reschedule the timeout. Eventually, either the device will return, or devloss_tmo will fire, and when the timeout then fires, it will be handled normally. If the rport is not blocked, normal error handling continues.

Notes

This routine assumes no locks are held on entry.

void **fc_remove_host**(struct Scsi_Host * *shost*)
called to terminate any fc_transport-related elements for a scsi host.

Parameters

struct Scsi_Host * shost Which Scsi_Host

Description

This routine is expected to be called immediately preceding the a driver's call to *scsi_remove_host()*.

WARNING: A driver utilizing the fc_transport, which fails to call this routine prior to *scsi_remove_host()*, will leave dangling objects in /sys/class/fc_remote_ports. Access to any of these objects can result in a system crash !!!

Notes

This routine assumes no locks are held on entry.

struct fc_rport * **fc_remote_port_add**(struct Scsi_Host * *shost*, int *channel*, struct fc_rport_identifiers * *ids*)
notify fc transport of the existence of a remote FC port.

Parameters

struct Scsi_Host * shost scsi host the remote port is connected to.

int channel Channel on shost port connected to.

struct fc_rport_identifiers * ids The world wide names, fc address, and FC4 port roles for the remote port.

Description

The LLDD calls this routine to notify the transport of the existence of a remote port. The LLDD provides the unique identifiers (wwpn,wwn) of the port, it's FC address (port_id), and the FC4 roles that are active for the port.

For ports that are FCP targets (aka scsi targets), the FC transport maintains consistent target id bindings on behalf of the LLDD. A consistent target id binding is an assignment of a target id to a remote port identifier, which persists while the scsi host is attached. The remote port can disappear, then later reappear, and it's target id assignment remains the same. This allows for shifts in FC addressing (if binding by wwpn or wwnn) with no apparent changes to the scsi subsystem which is based on scsi host number and target id values. Bindings are only valid during the attachment of the scsi host. If the host detaches, then later re-attaches, target id bindings may change.

This routine is responsible for returning a remote port structure. The routine will search the list of remote ports it maintains internally on behalf of consistent target id mappings. If found, the remote port structure will be reused. Otherwise, a new remote port structure will be allocated.

Whenever a remote port is allocated, a new fc_remote_port class device is created.

Should not be called from interrupt context.

Notes

This routine assumes no locks are held on entry.

void **fc_remote_port_delete**(struct fc_rport * *rport*)
notifies the fc transport that a remote port is no longer in existence.

Parameters

struct fc_rport * rport The remote port that no longer exists

Description

The LLDD calls this routine to notify the transport that a remote port is no longer part of the topology. Note: Although a port may no longer be part of the topology, it may persist in the remote ports displayed by the fc_host. We do this under 2 conditions:

1. If the port was a scsi target, we delay its deletion by "blocking" it. This allows the port to temporarily disappear, then reappear without disrupting the SCSI device tree attached to it. During the "blocked" period the port will still exist.
2. If the port was a scsi target and disappears for longer than we expect, we'll delete the port and the tear down the SCSI device tree attached to it. However, we want to semi-persist the target id assigned to that port if it eventually does exist. The port structure will remain (although with minimal information) so that the target id bindings also remain.

If the remote port is not an FCP Target, it will be fully torn down and deallocated, including the fc_remote_port class device.

If the remote port is an FCP Target, the port will be placed in a temporary blocked state. From the LLDD's perspective, the rport no longer exists. From the SCSI midlayer's perspective, the SCSI target exists, but all sdevs on it are blocked from further I/O. The following is then expected.

> If the remote port does not return (signaled by a LLDD call to `fc_remote_port_add()`) within the dev_loss_tmo timeout, then the scsi target is removed - killing all outstanding i/o and removing the scsi devices attached to it. The port structure will be marked Not Present and be partially cleared, leaving only enough information to recognize the remote port relative to the scsi target id binding if it later appears. The port will remain as long as there is a valid binding (e.g. until the user changes the binding type or unloads the scsi host with the binding).
>
> If the remote port returns within the dev_loss_tmo value (and matches according to the target id binding type), the port structure will be reused. If it is no longer a SCSI target, the target will be torn down. If it continues to be a SCSI target, then the target will be unblocked (allowing i/o to be resumed), and a scan will be activated to ensure that all luns are detected.

Called from normal process context only - cannot be called from interrupt.

Notes

> This routine assumes no locks are held on entry.

void **fc_remote_port_rolechg**(struct fc_rport * *rport*, u32 *roles*)
> notifies the fc transport that the roles on a remote may have changed.

Parameters

struct fc_rport * rport The remote port that changed.

u32 roles New roles for this port.

Description

The LLDD calls this routine to notify the transport that the roles on a remote port may have changed. The largest effect of this is if a port now becomes a FCP Target, it must be allocated a scsi target id. If the port is no longer a FCP target, any scsi target id value assigned to it will persist in case the role changes back to include FCP Target. No changes in the scsi midlayer will be invoked if the role changes (in the expectation that the role will be resumed. If it doesn't normal error processing will take place).

Should not be called from interrupt context.

Notes

> This routine assumes no locks are held on entry.

int **fc_block_rport**(struct fc_rport * *rport*)
> Block SCSI eh thread for blocked fc_rport.

Parameters

struct fc_rport * rport Remote port that scsi_eh is trying to recover.

Description

This routine can be called from a FC LLD scsi_eh callback. It blocks the scsi_eh thread until the fc_rport leaves the FC_PORTSTATE_BLOCKED, or the fast_io_fail_tmo fires. This is necessary to avoid the scsi_eh failing recovery actions for blocked rports which would lead to offlined SCSI devices.

Return

0 if the fc_rport left the state FC_PORTSTATE_BLOCKED. FAST_IO_FAIL if the fast_io_fail_tmo fired, this should be passed back to scsi_eh.

int **`fc_block_scsi_eh`**(struct scsi_cmnd * *cmnd*)
Block SCSI eh thread for blocked fc_rport

Parameters

`struct scsi_cmnd * cmnd` SCSI command that scsi_eh is trying to recover

Description

This routine can be called from a FC LLD scsi_eh callback. It blocks the scsi_eh thread until the fc_rport leaves the FC_PORTSTATE_BLOCKED, or the fast_io_fail_tmo fires. This is necessary to avoid the scsi_eh failing recovery actions for blocked rports which would lead to offlined SCSI devices.

Return

0 if the fc_rport left the state FC_PORTSTATE_BLOCKED. FAST_IO_FAIL if the fast_io_fail_tmo fired, this should be passed back to scsi_eh.

struct fc_vport * **`fc_vport_create`**(struct Scsi_Host * *shost*, int *channel*, struct fc_vport_identifiers * *ids*)
Admin App or LLDD requests creation of a vport

Parameters

`struct Scsi_Host * shost` scsi host the virtual port is connected to.

`int channel` channel on shost port connected to.

`struct fc_vport_identifiers * ids` The world wide names, FC4 port roles, etc for the virtual port.

Notes

This routine assumes no locks are held on entry.

int **`fc_vport_terminate`**(struct fc_vport * *vport*)
Admin App or LLDD requests termination of a vport

Parameters

`struct fc_vport * vport` fc_vport to be terminated

Description

Calls the LLDD `vport_delete()` function, then deallocates and removes the vport from the shost and object tree.

Notes

This routine assumes no locks are held on entry.

iSCSI transport class

The file drivers/scsi/scsi_transport_iscsi.c defines transport attributes for the iSCSI class, which sends SCSI packets over TCP/IP connections.

struct iscsi_bus_flash_session * **`iscsi_create_flashnode_sess`**(struct Scsi_Host * *shost*, int *index*, struct iscsi_transport * *transport*, int *dd_size*)
Add flashnode session entry in sysfs

Parameters

struct Scsi_Host * shost pointer to host data

int index index of flashnode to add in sysfs

struct iscsi_transport * transport pointer to transport data

int dd_size total size to allocate

Description

Adds a sysfs entry for the flashnode session attributes

Return

pointer to allocated flashnode sess on success NULL on failure

struct iscsi_bus_flash_conn * **iscsi_create_flashnode_conn**(struct Scsi_Host * *shost*, struct iscsi_bus_flash_session * *fnode_sess*, struct iscsi_transport * *transport*, int *dd_size*)

Add flashnode conn entry in sysfs

Parameters

struct Scsi_Host * shost pointer to host data

struct iscsi_bus_flash_session * fnode_sess pointer to the parent flashnode session entry

struct iscsi_transport * transport pointer to transport data

int dd_size total size to allocate

Description

Adds a sysfs entry for the flashnode connection attributes

Return

pointer to allocated flashnode conn on success NULL on failure

struct *device* * **iscsi_find_flashnode_sess**(struct Scsi_Host * *shost*, void * *data*, int (*fn) (struct *device* **dev*, void **data*)

finds flashnode session entry

Parameters

struct Scsi_Host * shost pointer to host data

void * data pointer to data containing value to use for comparison

int (*)(struct device *dev, void *data) fn function pointer that does actual comparison

Description

Finds the flashnode session object comparing the data passed using logic defined in passed function pointer

Return

pointer to found flashnode session device object on success NULL on failure

struct *device* * **iscsi_find_flashnode_conn**(struct iscsi_bus_flash_session * *fnode_sess*)

finds flashnode connection entry

Parameters

struct iscsi_bus_flash_session * fnode_sess pointer to parent flashnode session entry

Description

Finds the flashnode connection object comparing the data passed using logic defined in passed function pointer

Return

pointer to found flashnode connection device object on success NULL on failure

void **iscsi_destroy_flashnode_sess**(struct iscsi_bus_flash_session * *fnode_sess*)
destroy flashnode session entry

Parameters

struct iscsi_bus_flash_session * fnode_sess pointer to flashnode session entry to be destroyed

Description

Deletes the flashnode session entry and all children flashnode connection entries from sysfs

void **iscsi_destroy_all_flashnode**(struct Scsi_Host * *shost*)
destroy all flashnode session entries

Parameters

struct Scsi_Host * shost pointer to host data

Description

Destroys all the flashnode session entries and all corresponding children flashnode connection entries from sysfs

int **iscsi_scan_finished**(struct Scsi_Host * *shost*, unsigned long *time*)
helper to report when running scans are done

Parameters

struct Scsi_Host * shost scsi host

unsigned long time scan run time

Description

This function can be used by drives like qla4xxx to report to the scsi layer when the scans it kicked off at module load time are done.

int **iscsi_block_scsi_eh**(struct scsi_cmnd * *cmd*)
block scsi eh until session state has transistioned

Parameters

struct scsi_cmnd * cmd scsi cmd passed to scsi eh handler

Description

If the session is down this function will wait for the recovery timer to fire or for the session to be logged back in. If the recovery timer fires then FAST_IO_FAIL is returned. The caller should pass this error value to the scsi eh.

void **iscsi_unblock_session**(struct iscsi_cls_session * *session*)
set a session as logged in and start IO.

Parameters

struct iscsi_cls_session * session iscsi session

Description

Mark a session as ready to accept IO.

struct iscsi_cls_session * **iscsi_create_session**(struct Scsi_Host * *shost*, struct iscsi_transport * *transport*, int *dd_size*, unsigned int *target_id*)
create iscsi class session

Parameters

struct Scsi_Host * shost scsi host

struct iscsi_transport * transport iscsi transport

int dd_size private driver data size

unsigned int target_id which target

Description

This can be called from a LLD or iscsi_transport.

struct iscsi_cls_conn * **iscsi_create_conn**(struct iscsi_cls_session * *session*, int *dd_size*, uint32_t *cid*)
create iscsi class connection

Parameters

struct iscsi_cls_session * session iscsi cls session

int dd_size private driver data size

uint32_t cid connection id

Description

This can be called from a LLD or iscsi_transport. The connection is child of the session so cid must be unique for all connections on the session.

Since we do not support MCS, cid will normally be zero. In some cases for software iscsi we could be trying to preallocate a connection struct in which case there could be two connection structs and cid would be non-zero.

int **iscsi_destroy_conn**(struct iscsi_cls_conn * *conn*)
destroy iscsi class connection

Parameters

struct iscsi_cls_conn * conn iscsi cls session

Description

This can be called from a LLD or iscsi_transport.

int **iscsi_session_event**(struct iscsi_cls_session * *session*, enum iscsi_uevent_e *event*)
send session destr. completion event

Parameters

struct iscsi_cls_session * session iscsi class session

enum iscsi_uevent_e event type of event

Serial Attached SCSI (SAS) transport class

The file drivers/scsi/scsi_transport_sas.c defines transport attributes for Serial Attached SCSI, a variant of SATA aimed at large high-end systems.

The SAS transport class contains common code to deal with SAS HBAs, an aproximated representation of SAS topologies in the driver model, and various sysfs attributes to expose these topologies and management interfaces to userspace.

In addition to the basic SCSI core objects this transport class introduces two additional intermediate objects: The SAS PHY as represented by struct sas_phy defines an "outgoing" PHY on a SAS HBA or Expander, and the SAS remote PHY represented by struct sas_rphy defines an "incoming" PHY on a SAS Expander or end device. Note that this is purely a software concept, the underlying hardware for a PHY and a remote PHY is the exactly the same.

There is no concept of a SAS port in this code, users can see what PHYs form a wide port based on the port_identifier attribute, which is the same for all PHYs in a port.

void **sas_remove_children**(struct device * *dev*)
tear down a devices SAS data structures

Parameters

struct device * dev device belonging to the sas object

Description

Removes all SAS PHYs and remote PHYs for a given object

void **sas_remove_host**(struct Scsi_Host * *shost*)
tear down a Scsi_Host's SAS data structures

Parameters

struct Scsi_Host * shost Scsi Host that is torn down

Description

Removes all SAS PHYs and remote PHYs for a given Scsi_Host and remove the Scsi_Host as well.

Note

Do not call *scsi_remove_host()* on the Scsi_Host any more, as it is already removed.

u64 **sas_get_address**(struct scsi_device * *sdev*)
return the SAS address of the device

Parameters

struct scsi_device * sdev scsi device

Description

Returns the SAS address of the scsi device

unsigned int **sas_tlr_supported**(struct scsi_device * *sdev*)
checking TLR bit in vpd 0x90

Parameters

struct scsi_device * sdev scsi device struct

Description

Check Transport Layer Retries are supported or not. If vpd page 0x90 is present, TRL is supported.

void **sas_disable_tlr**(struct scsi_device * *sdev*)
setting TLR flags

Parameters

struct scsi_device * sdev scsi device struct

Description

Seting tlr_enabled flag to 0.

void **sas_enable_tlr**(struct scsi_device * *sdev*)
setting TLR flags

Parameters

struct scsi_device * sdev scsi device struct

Description

Seting tlr_enabled flag 1.

struct sas_phy * **sas_phy_alloc**(struct device * *parent*, int *number*)
allocates and initialize a SAS PHY structure

Parameters

struct device * parent Parent device

int number Phy index

Description

Allocates an SAS PHY structure. It will be added in the device tree below the device specified by **parent**, which has to be either a Scsi_Host or sas_rphy.

Return

SAS PHY allocated or `NULL` if the allocation failed.

int **sas_phy_add**(struct sas_phy * *phy*)
add a SAS PHY to the device hierarchy

Parameters

struct sas_phy * phy The PHY to be added

Description

Publishes a SAS PHY to the rest of the system.

void **sas_phy_free**(struct sas_phy * *phy*)
free a SAS PHY

Parameters

struct sas_phy * phy SAS PHY to free

Description

Frees the specified SAS PHY.

Note

This function must only be called on a PHY that has not successfully been added using *sas_phy_add()*.

void **sas_phy_delete**(struct sas_phy * *phy*)
remove SAS PHY

Parameters

struct sas_phy * phy SAS PHY to remove

Description

Removes the specified SAS PHY. If the SAS PHY has an associated remote PHY it is removed before.

int **scsi_is_sas_phy**(const struct *device* * *dev*)
check if a struct device represents a SAS PHY

Parameters

const struct device * dev device to check

Return

1 if the device represents a SAS PHY, `0` else

int **sas_port_add**(struct sas_port * *port*)
add a SAS port to the device hierarchy

Parameters

struct sas_port * port port to be added

Description

publishes a port to the rest of the system

void **sas_port_free**(struct sas_port * *port*)
free a SAS PORT

Parameters

struct sas_port * port SAS PORT to free

Description

Frees the specified SAS PORT.

Note

This function must only be called on a PORT that has not successfully been added using *sas_port_add()*.

void **sas_port_delete**(struct sas_port * *port*)
remove SAS PORT

Parameters

struct sas_port * port SAS PORT to remove

Description

Removes the specified SAS PORT. If the SAS PORT has an associated phys, unlink them from the port as well.

int **scsi_is_sas_port**(const struct *device* * *dev*)
check if a struct device represents a SAS port

Parameters

const struct device * dev device to check

Return

1 if the device represents a SAS Port, 0 else

struct sas_phy * **sas_port_get_phy**(struct sas_port * *port*)
try to take a reference on a port member

Parameters

struct sas_port * port port to check

void **sas_port_add_phy**(struct sas_port * *port*, struct sas_phy * *phy*)
add another phy to a port to form a wide port

Parameters

struct sas_port * port port to add the phy to

struct sas_phy * phy phy to add

Description

When a port is initially created, it is empty (has no phys). All ports must have at least one phy to operated, and all wide ports must have at least two. The current code makes no difference between ports and wide ports, but the only object that can be connected to a remote device is a port, so ports must be formed on all devices with phys if they're connected to anything.

void **sas_port_delete_phy**(struct sas_port * *port*, struct sas_phy * *phy*)
remove a phy from a port or wide port

Parameters

struct sas_port * port port to remove the phy from

struct sas_phy * phy phy to remove

Description

This operation is used for tearing down ports again. It must be done to every port or wide port before calling sas_port_delete.

struct sas_rphy * **sas_end_device_alloc**(struct sas_port * *parent*)
allocate an rphy for an end device

Parameters

struct sas_port * parent which port

Description

Allocates an SAS remote PHY structure, connected to **parent**.

Return

SAS PHY allocated or NULL if the allocation failed.

struct sas_rphy * **sas_expander_alloc**(struct sas_port * *parent*, enum sas_device_type *type*)
allocate an rphy for an end device

Parameters

struct sas_port * parent which port

enum sas_device_type type SAS_EDGE_EXPANDER_DEVICE or SAS_FANOUT_EXPANDER_DEVICE

Description

Allocates an SAS remote PHY structure, connected to **parent**.

Return

SAS PHY allocated or NULL if the allocation failed.

int **sas_rphy_add**(struct sas_rphy * *rphy*)
add a SAS remote PHY to the device hierarchy

Parameters

struct sas_rphy * rphy The remote PHY to be added

Description

Publishes a SAS remote PHY to the rest of the system.

void **sas_rphy_free**(struct sas_rphy * *rphy*)
free a SAS remote PHY

Parameters

struct sas_rphy * rphy SAS remote PHY to free

Description

Frees the specified SAS remote PHY.

Note

This function must only be called on a remote PHY that has not successfully been added using *sas_rphy_add()* (or has been *sas_rphy_remove()*'d)

void **sas_rphy_delete**(struct sas_rphy * *rphy*)
remove and free SAS remote PHY

Parameters

struct sas_rphy * rphy SAS remote PHY to remove and free

Description

Removes the specified SAS remote PHY and frees it.

void **sas_rphy_unlink**(struct sas_rphy * *rphy*)
unlink SAS remote PHY

Parameters

struct sas_rphy * rphy SAS remote phy to unlink from its parent port

Description

Removes port reference to an rphy

void **sas_rphy_remove**(struct sas_rphy * *rphy*)
remove SAS remote PHY

Parameters

struct sas_rphy * rphy SAS remote phy to remove

Description

Removes the specified SAS remote PHY.

int **scsi_is_sas_rphy**(const struct *device* * *dev*)
check if a struct device represents a SAS remote PHY

Parameters

const struct device * dev device to check

Return

1 if the device represents a SAS remote PHY, `0` else

struct scsi_transport_template * **sas_attach_transport**(struct sas_function_template * *ft*)
instantiate SAS transport template

Parameters

struct sas_function_template * ft SAS transport class function template

void **sas_release_transport**(struct scsi_transport_template * *t*)
release SAS transport template instance

Parameters

struct scsi_transport_template * t transport template instance

SATA transport class

The SATA transport is handled by libata, which has its own book of documentation in this directory.

Parallel SCSI (SPI) transport class

The file drivers/scsi/scsi_transport_spi.c defines transport attributes for traditional (fast/wide/ultra) SCSI busses.

void **spi_schedule_dv_device**(struct scsi_device * *sdev*)
schedule domain validation to occur on the device

Parameters

struct scsi_device * sdev The device to validate

Description

Identical to `spi_dv_device()` above, except that the DV will be scheduled to occur in a workqueue later. All memory allocations are atomic, so may be called from any context including those holding SCSI locks.

void **spi_display_xfer_agreement**(struct scsi_target * *starget*)
Print the current target transfer agreement

Parameters

struct scsi_target * starget The target for which to display the agreement

Description

Each SPI port is required to maintain a transfer agreement for each other port on the bus. This function prints a one-line summary of the current agreement; more detailed information is available in sysfs.

int **spi_populate_tag_msg**(unsigned char * *msg*, struct scsi_cmnd * *cmd*)
place a tag message in a buffer

Parameters

unsigned char * msg pointer to the area to place the tag

struct scsi_cmnd * cmd pointer to the scsi command for the tag

Notes

designed to create the correct type of tag message for the particular request. Returns the size of the tag message. May return 0 if TCQ is disabled for this device.

SCSI RDMA (SRP) transport class

The file drivers/scsi/scsi_transport_srp.c defines transport attributes for SCSI over Remote Direct Memory Access.

int **srp_tmo_valid**(int *reconnect_delay*, int *fast_io_fail_tmo*, long *dev_loss_tmo*)
check timeout combination validity

Parameters

int reconnect_delay Reconnect delay in seconds.

int fast_io_fail_tmo Fast I/O fail timeout in seconds.

long dev_loss_tmo Device loss timeout in seconds.

Description

The combination of the timeout parameters must be such that SCSI commands are finished in a reasonable time. Hence do not allow the fast I/O fail timeout to exceed SCSI_DEVICE_BLOCK_MAX_TIMEOUT nor allow dev_loss_tmo to exceed that limit if failing I/O fast has been disabled. Furthermore, these parameters must be such that multipath can detect failed paths timely. Hence do not allow all three parameters to be disabled simultaneously.

void **srp_start_tl_fail_timers**(struct srp_rport * *rport*)
start the transport layer failure timers

Parameters

struct srp_rport * rport SRP target port.

Description

Start the transport layer fast I/O failure and device loss timers. Do not modify a timer that was already started.

int **srp_reconnect_rport**(struct srp_rport * *rport*)
reconnect to an SRP target port

Parameters

struct srp_rport * rport SRP target port.

Description

Blocks SCSI command queueing before invoking `reconnect()` such that `queuecommand()` won't be invoked concurrently with `reconnect()` from outside the SCSI EH. This is important since a `reconnect()` implementation may reallocate resources needed by `queuecommand()`.

Notes

- This function neither waits until outstanding requests have finished nor tries to abort these. It is the responsibility of the `reconnect()` function to finish outstanding commands before reconnecting to the target port.

- It is the responsibility of the caller to ensure that the resources reallocated by the `reconnect()` function won't be used while this function is in progress. One possible strategy is to invoke this function from the context of the SCSI EH thread only. Another possible strategy is to lock the rport mutex inside each SCSI LLD callback that can be invoked by the SCSI EH (the scsi_host_template.eh_*() functions and also the scsi_host_template.:c:func:*queuecommand()* function).

enum blk_eh_timer_return **`srp_timed_out`**(struct scsi_cmnd * *scmd*)
SRP transport intercept of the SCSI timeout EH

Parameters

`struct scsi_cmnd * scmd` SCSI command.

Description

If a timeout occurs while an rport is in the blocked state, ask the SCSI EH to continue waiting (BLK_EH_RESET_TIMER). Otherwise let the SCSI core handle the timeout (BLK_EH_NOT_HANDLED).

Note

This function is called from soft-IRQ context and with the request queue lock held.

void **`srp_rport_get`**(struct srp_rport * *rport*)
increment rport reference count

Parameters

`struct srp_rport * rport` SRP target port.

void **`srp_rport_put`**(struct srp_rport * *rport*)
decrement rport reference count

Parameters

`struct srp_rport * rport` SRP target port.

struct srp_rport * **`srp_rport_add`**(struct Scsi_Host * *shost*, struct srp_rport_identifiers * *ids*)
add a SRP remote port to the device hierarchy

Parameters

`struct Scsi_Host * shost` scsi host the remote port is connected to.

`struct srp_rport_identifiers * ids` The port id for the remote port.

Description

Publishes a port to the rest of the system.

void **`srp_rport_del`**(struct srp_rport * *rport*)
remove a SRP remote port

Parameters

`struct srp_rport * rport` SRP remote port to remove

Description

Removes the specified SRP remote port.

void **`srp_remove_host`**(struct Scsi_Host * *shost*)
tear down a Scsi_Host's SRP data structures

Parameters

`struct Scsi_Host * shost` Scsi Host that is torn down

Description

Removes all SRP remote ports for a given Scsi_Host. Must be called just before scsi_remove_host for SRP HBAs.

void **srp_stop_rport_timers**(struct srp_rport * *rport*)
stop the transport layer recovery timers

Parameters

struct srp_rport * rport SRP remote port for which to stop the timers.

Description

Must be called after *srp_remove_host()* and *scsi_remove_host()*. The caller must hold a reference on the rport (rport->dev) and on the SCSI host (rport->dev.parent).

struct scsi_transport_template * **srp_attach_transport**(struct srp_function_template * *ft*)
instantiate SRP transport template

Parameters

struct srp_function_template * ft SRP transport class function template

void **srp_release_transport**(struct scsi_transport_template * *t*)
release SRP transport template instance

Parameters

struct scsi_transport_template * t transport template instance

SCSI lower layer

Host Bus Adapter transport types

Many modern device controllers use the SCSI command set as a protocol to communicate with their devices through many different types of physical connections.

In SCSI language a bus capable of carrying SCSI commands is called a "transport", and a controller connecting to such a bus is called a "host bus adapter" (HBA).

Debug transport

The file drivers/scsi/scsi_debug.c simulates a host adapter with a variable number of disks (or disk like devices) attached, sharing a common amount of RAM. Does a lot of checking to make sure that we are not getting blocks mixed up, and panics the kernel if anything out of the ordinary is seen.

To be more realistic, the simulated devices have the transport attributes of SAS disks.

For documentation see http://sg.danny.cz/sg/sdebug26.html

todo

Parallel (fast/wide/ultra) SCSI, USB, SATA, SAS, Fibre Channel, FireWire, ATAPI devices, Infiniband, I2O, iSCSI, Parallel ports, netlink...

CHAPTER

TWENTYONE

LIBATA DEVELOPER'S GUIDE

Author Jeff Garzik

Introduction

libATA is a library used inside the Linux kernel to support ATA host controllers and devices. libATA provides an ATA driver API, class transports for ATA and ATAPI devices, and SCSI<->ATA translation for ATA devices according to the T10 SAT specification.

This Guide documents the libATA driver API, library functions, library internals, and a couple sample ATA low-level drivers.

libata Driver API

`struct ata_port_operations` is defined for every low-level libata hardware driver, and it controls how the low-level driver interfaces with the ATA and SCSI layers.

FIS-based drivers will hook into the system with `->qc_prep()` and `->qc_issue()` high-level hooks. Hardware which behaves in a manner similar to PCI IDE hardware may utilize several generic helpers, defining at a bare minimum the bus I/O addresses of the ATA shadow register blocks.

`struct ata_port_operations`

Disable ATA port

```
void (*port_disable) (struct ata_port *);
```

Called from *`ata_bus_probe()`* error path, as well as when unregistering from the SCSI module (rmmod, hot unplug). This function should do whatever needs to be done to take the port out of use. In most cases, `ata_port_disable()` can be used as this hook.

Called from *`ata_bus_probe()`* on a failed probe. Called from `ata_scsi_release()`.

Post-IDENTIFY device configuration

```
void (*dev_config) (struct ata_port *, struct ata_device *);
```

Called after IDENTIFY [PACKET] DEVICE is issued to each device found. Typically used to apply device-specific fixups prior to issue of SET FEATURES - XFER MODE, and prior to operation.

This entry may be specified as NULL in ata_port_operations.

Set PIO/DMA mode

```
void (*set_piomode) (struct ata_port *, struct ata_device *);
void (*set_dmamode) (struct ata_port *, struct ata_device *);
void (*post_set_mode) (struct ata_port *);
unsigned int (*mode_filter) (struct ata_port *, struct ata_device *, unsigned int);
```

Hooks called prior to the issue of SET FEATURES - XFER MODE command. The optional `->mode_filter()` hook is called when libata has built a mask of the possible modes. This is passed to the `->mode_filter()` function which should return a mask of valid modes after filtering those unsuitable due to hardware limits. It is not valid to use this interface to add modes.

`dev->pio_mode` and `dev->dma_mode` are guaranteed to be valid when `->set_piomode()` and when `->set_dmamode()` is called. The timings for any other drive sharing the cable will also be valid at this point. That is the library records the decisions for the modes of each drive on a channel before it attempts to set any of them.

`->post_set_mode()` is called unconditionally, after the SET FEATURES - XFER MODE command completes successfully.

`->set_piomode()` is always called (if present), but `->set_dma_mode()` is only called if DMA is possible.

Taskfile read/write

```
void (*sff_tf_load) (struct ata_port *ap, struct ata_taskfile *tf);
void (*sff_tf_read) (struct ata_port *ap, struct ata_taskfile *tf);
```

`->tf_load()` is called to load the given taskfile into hardware registers / DMA buffers. `->tf_read()` is called to read the hardware registers / DMA buffers, to obtain the current set of taskfile register values. Most drivers for taskfile-based hardware (PIO or MMIO) use `ata_sff_tf_load()` and `ata_sff_tf_read()` for these hooks.

PIO data read/write

```
void (*sff_data_xfer) (struct ata_device *, unsigned char *, unsigned int, int);
```

All bmdma-style drivers must implement this hook. This is the low-level operation that actually copies the data bytes during a PIO data transfer. Typically the driver will choose one of `ata_sff_data_xfer_noirq()`, `ata_sff_data_xfer()`, or `ata_sff_data_xfer32()`.

ATA command execute

```
void (*sff_exec_command)(struct ata_port *ap, struct ata_taskfile *tf);
```

causes an ATA command, previously loaded with `->tf_load()`, to be initiated in hardware. Most drivers for taskfile-based hardware use `ata_sff_exec_command()` for this hook.

Per-cmd ATAPI DMA capabilities filter

```
int (*check_atapi_dma) (struct ata_queued_cmd *qc);
```

Allow low-level driver to filter ATA PACKET commands, returning a status indicating whether or not it is OK to use DMA for the supplied PACKET command.

This hook may be specified as NULL, in which case libata will assume that atapi dma can be supported.

Read specific ATA shadow registers

```
u8   (*sff_check_status)(struct ata_port *ap);
u8   (*sff_check_altstatus)(struct ata_port *ap);
```

Reads the Status/AltStatus ATA shadow register from hardware. On some hardware, reading the Status register has the side effect of clearing the interrupt condition. Most drivers for taskfile-based hardware use `ata_sff_check_status()` for this hook.

Write specific ATA shadow register

```
void (*sff_set_devctl)(struct ata_port *ap, u8 ctl);
```

Write the device control ATA shadow register to the hardware. Most drivers don't need to define this.

Select ATA device on bus

```
void (*sff_dev_select)(struct ata_port *ap, unsigned int device);
```

Issues the low-level hardware command(s) that causes one of N hardware devices to be considered 'selected' (active and available for use) on the ATA bus. This generally has no meaning on FIS-based devices.

Most drivers for taskfile-based hardware use `ata_sff_dev_select()` for this hook.

Private tuning method

```
void (*set_mode) (struct ata_port *ap);
```

By default libata performs drive and controller tuning in accordance with the ATA timing rules and also applies blacklists and cable limits. Some controllers need special handling and have custom tuning rules, typically raid controllers that use ATA commands but do not actually do drive timing.

Warning

This hook should not be used to replace the standard controller tuning logic when a controller has quirks. Replacing the default tuning logic in that case would bypass handling for drive and bridge quirks that may be important to data reliability. If a controller needs to filter the mode selection it should use the mode_filter hook instead.

Control PCI IDE BMDMA engine

```
void (*bmdma_setup) (struct ata_queued_cmd *qc);
void (*bmdma_start) (struct ata_queued_cmd *qc);
void (*bmdma_stop) (struct ata_port *ap);
u8   (*bmdma_status) (struct ata_port *ap);
```

When setting up an IDE BMDMA transaction, these hooks arm (`->bmdma_setup`), fire (`->bmdma_start`), and halt (`->bmdma_stop`) the hardware's DMA engine. `->bmdma_status` is used to read the standard PCI IDE DMA Status register.

These hooks are typically either no-ops, or simply not implemented, in FIS-based drivers.

Most legacy IDE drivers use `ata_bmdma_setup()` for the `bmdma_setup()` hook. `ata_bmdma_setup()` will write the pointer to the PRD table to the IDE PRD Table Address register, enable DMA in the DMA Command register, and call `exec_command()` to begin the transfer.

Most legacy IDE drivers use `ata_bmdma_start()` for the `bmdma_start()` hook. `ata_bmdma_start()` will write the ATA_DMA_START flag to the DMA Command register.

Many legacy IDE drivers use `ata_bmdma_stop()` for the `bmdma_stop()` hook. `ata_bmdma_stop()` clears the ATA_DMA_START flag in the DMA command register.

Many legacy IDE drivers use `ata_bmdma_status()` as the `bmdma_status()` hook.

High-level taskfile hooks

```
void (*qc_prep) (struct ata_queued_cmd *qc);
int (*qc_issue) (struct ata_queued_cmd *qc);
```

Higher-level hooks, these two hooks can potentially supercede several of the above taskfile/DMA engine hooks. `->qc_prep` is called after the buffers have been DMA-mapped, and is typically used to populate the hardware's DMA scatter-gather table. Most drivers use the standard `ata_qc_prep()` helper function, but more advanced drivers roll their own.

`->qc_issue` is used to make a command active, once the hardware and S/G tables have been prepared. IDE BMDMA drivers use the helper function `ata_qc_issue_prot()` for taskfile protocol-based dispatch. More advanced drivers implement their own `->qc_issue`.

`ata_qc_issue_prot()` calls `->tf_load()`, `->bmdma_setup()`, and `->bmdma_start()` as necessary to initiate a transfer.

Exception and probe handling (EH)

```
void (*eng_timeout) (struct ata_port *ap);
void (*phy_reset) (struct ata_port *ap);
```

Deprecated. Use `->error_handler()` instead.

```
void (*freeze) (struct ata_port *ap);
void (*thaw) (struct ata_port *ap);
```

`ata_port_freeze()` is called when HSM violations or some other condition disrupts normal operation of the port. A frozen port is not allowed to perform any operation until the port is thawed, which usually follows a successful reset.

The optional `->freeze()` callback can be used for freezing the port hardware-wise (e.g. mask interrupt and stop DMA engine). If a port cannot be frozen hardware-wise, the interrupt handler must ack and clear interrupts unconditionally while the port is frozen.

The optional `->thaw()` callback is called to perform the opposite of `->freeze()`: prepare the port for normal operation once again. Unmask interrupts, start DMA engine, etc.

```
void (*error_handler) (struct ata_port *ap);
```

`->error_handler()` is a driver's hook into probe, hotplug, and recovery and other exceptional conditions. The primary responsibility of an implementation is to call *`ata_do_eh()`* or `ata_bmdma_drive_eh()` with a set of EH hooks as arguments:

'prereset' hook (may be NULL) is called during an EH reset, before any other actions are taken.

'postreset' hook (may be NULL) is called after the EH reset is performed. Based on existing conditions, severity of the problem, and hardware capabilities,

Either 'softreset' (may be NULL) or 'hardreset' (may be NULL) will be called to perform the low-level EH reset.

```
void (*post_internal_cmd) (struct ata_queued_cmd *qc);
```

Perform any hardware-specific actions necessary to finish processing after executing a probe-time or EH-time command via *`ata_exec_internal()`*.

Hardware interrupt handling

```
irqreturn_t (*irq_handler)(int, void *, struct pt_regs *);
void (*irq_clear) (struct ata_port *);
```

`->irq_handler` is the interrupt handling routine registered with the system, by libata. `->irq_clear` is called during probe just before the interrupt handler is registered, to be sure hardware is quiet.

The second argument, dev_instance, should be cast to a pointer to `struct ata_host_set`.

Most legacy IDE drivers use `ata_sff_interrupt()` for the irq_handler hook, which scans all ports in the host_set, determines which queued command was active (if any), and calls ata_sff_host_intr(ap,qc).

Most legacy IDE drivers use `ata_sff_irq_clear()` for the `irq_clear()` hook, which simply clears the interrupt and error flags in the DMA status register.

SATA phy read/write

```
int (*scr_read) (struct ata_port *ap, unsigned int sc_reg,
         u32 *val);
int (*scr_write) (struct ata_port *ap, unsigned int sc_reg,
                   u32 val);
```

Read and write standard SATA phy registers. Currently only used if ->`phy_reset` hook called the `sata_phy_reset()` helper function. sc_reg is one of SCR_STATUS, SCR_CONTROL, SCR_ERROR, or SCR_ACTIVE.

Init and shutdown

```
int (*port_start) (struct ata_port *ap);
void (*port_stop) (struct ata_port *ap);
void (*host_stop) (struct ata_host_set *host_set);
```

`->port_start()` is called just after the data structures for each port are initialized. Typically this is used to alloc per-port DMA buffers / tables / rings, enable DMA engines, and similar tasks. Some drivers also use this entry point as a chance to allocate driver-private memory for `ap->private_data`.

Many drivers use `ata_port_start()` as this hook or call it from their own `port_start()` hooks. `ata_port_start()` allocates space for a legacy IDE PRD table and returns.

`->port_stop()` is called after `->host_stop()`. Its sole function is to release DMA/memory resources, now that they are no longer actively being used. Many drivers also free driver-private data from port at this time.

`->host_stop()` is called after all `->port_stop()` calls have completed. The hook must finalize hardware shutdown, release DMA and other resources, etc. This hook may be specified as NULL, in which case it is not called.

Error handling

This chapter describes how errors are handled under libata. Readers are advised to read SCSI EH (Documentation/scsi/scsi_eh.txt) and ATA exceptions doc first.

Origins of commands

In libata, a command is represented with `struct ata_queued_cmd` or qc. qc's are preallocated during port initialization and repetitively used for command executions. Currently only one qc is allocated per port but yet-to-be-merged NCQ branch allocates one for each tag and maps each qc to NCQ tag 1-to-1.

libata commands can originate from two sources - libata itself and SCSI midlayer. libata internal commands are used for initialization and error handling. All normal blk requests and commands for SCSI emulation are passed as SCSI commands through queuecommand callback of SCSI host template.

How commands are issued

Internal commands First, qc is allocated and initialized using *ata_qc_new_init()*. Although *ata_qc_new_init()* doesn't implement any wait or retry mechanism when qc is not available, internal commands are currently issued only during initialization and error recovery, so no other command is active and allocation is guaranteed to succeed.

Once allocated qc's taskfile is initialized for the command to be executed. qc currently has two mechanisms to notify completion. One is via `qc->complete_fn()` callback and the other is completion `qc->waiting`. `qc->complete_fn()` callback is the asynchronous path used by normal SCSI translated commands and `qc->waiting` is the synchronous (issuer sleeps in process context) path used by internal commands.

Once initialization is complete, host_set lock is acquired and the qc is issued.

SCSI commands All libata drivers use *ata_scsi_queuecmd()* as `hostt`->queuecommand callback. scmds can either be simulated or translated. No qc is involved in processing a simulated scmd. The result is computed right away and the scmd is completed.

For a translated scmd, *ata_qc_new_init()* is invoked to allocate a qc and the scmd is translated into the qc. SCSI midlayer's completion notification function pointer is stored into `qc->scsidone`.

`qc->complete_fn()` callback is used for completion notification. ATA commands use `ata_scsi_qc_complete()` while ATAPI commands use `atapi_qc_complete()`. Both functions end up calling `qc->scsidone` to notify upper layer when the qc is finished. After translation is completed, the qc is issued with *ata_qc_issue()*.

Note that SCSI midlayer invokes hostt->queuecommand while holding host_set lock, so all above occur while holding host_set lock.

How commands are processed

Depending on which protocol and which controller are used, commands are processed differently. For the purpose of discussion, a controller which uses taskfile interface and all standard callbacks is assumed.

Currently 6 ATA command protocols are used. They can be sorted into the following four categories according to how they are processed.

ATA NO DATA or DMA ATA_PROT_NODATA and ATA_PROT_DMA fall into this category. These types of commands don't require any software intervention once issued. Device will raise interrupt on completion.

ATA PIO ATA_PROT_PIO is in this category. libata currently implements PIO with polling. ATA_NIEN bit is set to turn off interrupt and pio_task on ata_wq performs polling and IO.

ATAPI NODATA or DMA ATA_PROT_ATAPI_NODATA and ATA_PROT_ATAPI_DMA are in this category. packet_task is used to poll BSY bit after issuing PACKET command. Once BSY is turned off by the device, packet_task transfers CDB and hands off processing to interrupt handler.

ATAPI PIO ATA_PROT_ATAPI is in this category. ATA_NIEN bit is set and, as in ATAPI NODATA or DMA, packet_task submits cdb. However, after submitting cdb, further processing (data transfer) is handed off to pio_task.

How commands are completed

Once issued, all qc's are either completed with *ata_qc_complete()* or time out. For commands which are handled by interrupts, `ata_host_intr()` invokes *ata_qc_complete()*, and, for PIO tasks, pio_task invokes *ata_qc_complete()*. In error cases, packet_task may also complete commands.

ata_qc_complete() does the following.

1. DMA memory is unmapped.
2. ATA_QCFLAG_ACTIVE is cleared from qc->flags.
3. `qc->complete_fn()` callback is invoked. If the return value of the callback is not zero. Completion is short circuited and *ata_qc_complete()* returns.
4. `__ata_qc_complete()` is called, which does
   (a) `qc->flags` is cleared to zero.
   (b) `ap->active_tag` and `qc->tag` are poisoned.
   (c) `qc->waiting` is cleared & completed (in that order).
   (d) qc is deallocated by clearing appropriate bit in `ap->qactive`.

So, it basically notifies upper layer and deallocates qc. One exception is short-circuit path in #3 which is used by `atapi_qc_complete()`.

For all non-ATAPI commands, whether it fails or not, almost the same code path is taken and very little error handling takes place. A qc is completed with success status if it succeeded, with failed status otherwise.

However, failed ATAPI commands require more handling as REQUEST SENSE is needed to acquire sense data. If an ATAPI command fails, *ata_qc_complete()* is invoked with error status, which in turn invokes `atapi_qc_complete()` via `qc->complete_fn()` callback.

This makes `atapi_qc_complete()` set `scmd->result` to SAM_STAT_CHECK_CONDITION, complete the scmd and return 1. As the sense data is empty but `scmd->result` is CHECK CONDITION, SCSI midlayer will invoke EH for the scmd, and returning 1 makes *ata_qc_complete()* to return without deallocating the qc. This leads us to *ata_scsi_error()* with partially completed qc.

ata_scsi_error()

ata_scsi_error() is the current `transportt->eh_strategy_handler()` for libata. As discussed above, this will be entered in two cases - timeout and ATAPI error completion. This function calls low level libata driver's `eng_timeout()` callback, the standard callback for which is `ata_eng_timeout()`. It checks if a qc is active and calls `ata_qc_timeout()` on the qc if so. Actual error handling occurs in `ata_qc_timeout()`.

If EH is invoked for timeout, `ata_qc_timeout()` stops BMDMA and completes the qc. Note that as we're currently in EH, we cannot call scsi_done. As described in SCSI EH doc, a recovered scmd should be either retried with `scsi_queue_insert()` or finished with `scsi_finish_command()`. Here, we override `qc->scsidone` with `scsi_finish_command()` and calls *ata_qc_complete()*.

If EH is invoked due to a failed ATAPI qc, the qc here is completed but not deallocated. The purpose of this half-completion is to use the qc as place holder to make EH code reach this place. This is a bit hackish, but it works.

Once control reaches here, the qc is deallocated by invoking `__ata_qc_complete()` explicitly. Then, internal qc for REQUEST SENSE is issued. Once sense data is acquired, scmd is finished by directly invoking `scsi_finish_command()` on the scmd. Note that as we already have completed and deallocated the qc which was associated with the scmd, we don't need to/cannot call *ata_qc_complete()* again.

Problems with the current EH

- Error representation is too crude. Currently any and all error conditions are represented with ATA STATUS and ERROR registers. Errors which aren't ATA device errors are treated as ATA device errors by setting ATA_ERR bit. Better error descriptor which can properly represent ATA and other errors/exceptions is needed.

- When handling timeouts, no action is taken to make device forget about the timed out command and ready for new commands.
- EH handling via *ata_scsi_error()* is not properly protected from usual command processing. On EH entrance, the device is not in quiescent state. Timed out commands may succeed or fail any time. pio_task and atapi_task may still be running.
- Too weak error recovery. Devices / controllers causing HSM mismatch errors and other errors quite often require reset to return to known state. Also, advanced error handling is necessary to support features like NCQ and hotplug.
- ATA errors are directly handled in the interrupt handler and PIO errors in pio_task. This is problematic for advanced error handling for the following reasons.

  First, advanced error handling often requires context and internal qc execution.

  Second, even a simple failure (say, CRC error) needs information gathering and could trigger complex error handling (say, resetting & reconfiguring). Having multiple code paths to gather information, enter EH and trigger actions makes life painful.

  Third, scattered EH code makes implementing low level drivers difficult. Low level drivers override libata callbacks. If EH is scattered over several places, each affected callbacks should perform its part of error handling. This can be error prone and painful.

libata Library

struct ata_link * **ata_link_next**(struct ata_link * *link*, struct ata_port * *ap*, enum ata_link_iter_mode *mode*)

link iteration helper

Parameters

struct ata_link * link the previous link, NULL to start

struct ata_port * ap ATA port containing links to iterate

enum ata_link_iter_mode mode iteration mode, one of ATA_LITER_*

Description

LOCKING: Host lock or EH context.

Return

Pointer to the next link.

struct ata_device * **ata_dev_next**(struct ata_device * *dev*, struct ata_link * *link*, enum ata_dev_iter_mode *mode*)

device iteration helper

Parameters

struct ata_device * dev the previous device, NULL to start

struct ata_link * link ATA link containing devices to iterate

enum ata_dev_iter_mode mode iteration mode, one of ATA_DITER_*

Description

LOCKING: Host lock or EH context.

Return

Pointer to the next device.

int **atapi_cmd_type**(u8 *opcode*)

Determine ATAPI command type from SCSI opcode

Parameters

`u8 opcode` SCSI opcode

Description

Determine ATAPI command type from **opcode**.

LOCKING: None.

Return

ATAPI_{READ|WRITE|READ_CD|PASS_THRU|MISC}

void **`ata_tf_to_fis`**(const struct ata_taskfile * *tf*, u8 *pmp*, int *is_cmd*, u8 * *fis*)
Convert ATA taskfile to SATA FIS structure

Parameters

`const struct ata_taskfile * tf` Taskfile to convert

`u8 pmp` Port multiplier port

`int is_cmd` This FIS is for command

`u8 * fis` Buffer into which data will output

Description

Converts a standard ATA taskfile to a Serial ATA FIS structure (Register - Host to Device).

LOCKING: Inherited from caller.

void **`ata_tf_from_fis`**(const u8 * *fis*, struct ata_taskfile * *tf*)
Convert SATA FIS to ATA taskfile

Parameters

`const u8 * fis` Buffer from which data will be input

`struct ata_taskfile * tf` Taskfile to output

Description

Converts a serial ATA FIS structure to a standard ATA taskfile.

LOCKING: Inherited from caller.

unsigned long **`ata_pack_xfermask`**(unsigned long *pio_mask*, unsigned long *mwdma_mask*, unsigned long *udma_mask*)
Pack pio, mwdma and udma masks into xfer_mask

Parameters

`unsigned long pio_mask` pio_mask

`unsigned long mwdma_mask` mwdma_mask

`unsigned long udma_mask` udma_mask

Description

Pack **pio_mask**, **mwdma_mask** and **udma_mask** into a single unsigned int xfer_mask.

LOCKING: None.

Return

Packed xfer_mask.

void **`ata_unpack_xfermask`**(unsigned long *xfer_mask*, unsigned long * *pio_mask*, unsigned long * *mwdma_mask*, unsigned long * *udma_mask*)
Unpack xfer_mask into pio, mwdma and udma masks

Parameters

`unsigned long xfer_mask` xfer_mask to unpack

`unsigned long * pio_mask` resulting pio_mask

`unsigned long * mwdma_mask` resulting mwdma_mask

`unsigned long * udma_mask` resulting udma_mask

Description

Unpack **xfer_mask** into **pio_mask**, **mwdma_mask** and **udma_mask**. Any NULL destination masks will be ignored.

u8 **`ata_xfer_mask2mode`**(unsigned long *xfer_mask*)
Find matching XFER_* for the given xfer_mask

Parameters

`unsigned long xfer_mask` xfer_mask of interest

Description

Return matching XFER_* value for **xfer_mask**. Only the highest bit of **xfer_mask** is considered.

LOCKING: None.

Return

Matching XFER_* value, 0xff if no match found.

unsigned long **`ata_xfer_mode2mask`**(u8 *xfer_mode*)
Find matching xfer_mask for XFER_*

Parameters

`u8 xfer_mode` XFER_* of interest

Description

Return matching xfer_mask for **xfer_mode**.

LOCKING: None.

Return

Matching xfer_mask, 0 if no match found.

int **`ata_xfer_mode2shift`**(unsigned long *xfer_mode*)
Find matching xfer_shift for XFER_*

Parameters

`unsigned long xfer_mode` XFER_* of interest

Description

Return matching xfer_shift for **xfer_mode**.

LOCKING: None.

Return

Matching xfer_shift, -1 if no match found.

const char * **`ata_mode_string`**(unsigned long *xfer_mask*)
convert xfer_mask to string

Parameters

`unsigned long xfer_mask` mask of bits supported; only highest bit counts.

Description

Determine string which represents the highest speed (highest bit in **modemask**).

LOCKING: None.

Return

Constant C string representing highest speed listed in **mode_mask**, or the constant C string "<n/a>".

unsigned int **`ata_dev_classify`**(const struct ata_taskfile * *tf*)
determine device type based on ATA-spec signature

Parameters

`const struct ata_taskfile * tf` ATA taskfile register set for device to be identified

Description

Determine from taskfile register contents whether a device is ATA or ATAPI, as per "Signature and persistence" section of ATA/PI spec (volume 1, sect 5.14).

LOCKING: None.

Return

Device type, `ATA_DEV_ATA`, `ATA_DEV_ATAPI`, `ATA_DEV_PMP`, `ATA_DEV_ZAC`, or `ATA_DEV_UNKNOWN` the event of failure.

void **`ata_id_string`**(const u16 * *id*, unsigned char * *s*, unsigned int *ofs*, unsigned int *len*)
Convert IDENTIFY DEVICE page into string

Parameters

`const u16 * id` IDENTIFY DEVICE results we will examine

`unsigned char * s` string into which data is output

`unsigned int ofs` offset into identify device page

`unsigned int len` length of string to return. must be an even number.

Description

The strings in the IDENTIFY DEVICE page are broken up into 16-bit chunks. Run through the string, and output each 8-bit chunk linearly, regardless of platform.

LOCKING: caller.

void **`ata_id_c_string`**(const u16 * *id*, unsigned char * *s*, unsigned int *ofs*, unsigned int *len*)
Convert IDENTIFY DEVICE page into C string

Parameters

`const u16 * id` IDENTIFY DEVICE results we will examine

`unsigned char * s` string into which data is output

`unsigned int ofs` offset into identify device page

`unsigned int len` length of string to return. must be an odd number.

Description

This function is identical to ata_id_string except that it trims trailing spaces and terminates the resulting string with null. **len** must be actual maximum length (even number) + 1.

LOCKING: caller.

unsigned long **`ata_id_xfermask`**(const u16 * *id*)
Compute xfermask from the given IDENTIFY data

Parameters

`const u16 * id` IDENTIFY data to compute xfer mask from

Description

Compute the xfermask for this device. This is not as trivial as it seems if we must consider early devices correctly.

FIXME: pre IDE drive timing (do we care ?).

LOCKING: None.

Return

Computed xfermask

unsigned int **ata_pio_need_iordy**(const struct ata_device * *adev*)
check if iordy needed

Parameters

const struct ata_device * adev ATA device

Description

Check if the current speed of the device requires IORDY. Used by various controllers for chip configuration.

unsigned int **ata_do_dev_read_id**(struct ata_device * *dev*, struct ata_taskfile * *tf*, u16 * *id*)
default ID read method

Parameters

struct ata_device * dev device

struct ata_taskfile * tf proposed taskfile

u16 * id data buffer

Description

Issue the identify taskfile and hand back the buffer containing identify data. For some RAID controllers and for pre ATA devices this function is wrapped or replaced by the driver

int **ata_cable_40wire**(struct ata_port * *ap*)
return 40 wire cable type

Parameters

struct ata_port * ap port

Description

Helper method for drivers which want to hardwire 40 wire cable detection.

int **ata_cable_80wire**(struct ata_port * *ap*)
return 80 wire cable type

Parameters

struct ata_port * ap port

Description

Helper method for drivers which want to hardwire 80 wire cable detection.

int **ata_cable_unknown**(struct ata_port * *ap*)
return unknown PATA cable.

Parameters

struct ata_port * ap port

Description

Helper method for drivers which have no PATA cable detection.

int **ata_cable_ignore**(struct ata_port * *ap*)
return ignored PATA cable.

Parameters

`struct ata_port * ap` port

Description

Helper method for drivers which don't use cable type to limit transfer mode.

int **`ata_cable_sata`**(struct ata_port * *ap*)
return SATA cable type

Parameters

`struct ata_port * ap` port

Description

Helper method for drivers which have SATA cables

struct ata_device * **`ata_dev_pair`**(struct ata_device * *adev*)
return other device on cable

Parameters

`struct ata_device * adev` device

Description

Obtain the other device on the same cable, or if none is present NULL is returned

int **`sata_set_spd`**(struct ata_link * *link*)
set SATA spd according to spd limit

Parameters

`struct ata_link * link` Link to set SATA spd for

Description

Set SATA spd of **link** according to sata_spd_limit.

LOCKING: Inherited from caller.

Return

0 if spd doesn't need to be changed, 1 if spd has been changed. Negative errno if SCR registers are inaccessible.

u8 **`ata_timing_cycle2mode`**(unsigned int *xfer_shift*, int *cycle*)
find xfer mode for the specified cycle duration

Parameters

`unsigned int xfer_shift` ATA_SHIFT_* value for transfer type to examine.

`int cycle` cycle duration in ns

Description

Return matching xfer mode for **cycle**. The returned mode is of the transfer type specified by **xfer_shift**. If **cycle** is too slow for **xfer_shift**, 0xff is returned. If **cycle** is faster than the fastest known mode, the fasted mode is returned.

LOCKING: None.

Return

Matching xfer_mode, 0xff if no match found.

int **`ata_do_set_mode`**(struct ata_link * *link*, struct ata_device ** *r_failed_dev*)
Program timings and issue SET FEATURES - XFER

Parameters

`struct ata_link * link` link on which timings will be programmed

`struct ata_device ** r_failed_dev` out parameter for failed device

Description

Standard implementation of the function used to tune and set ATA device disk transfer mode (PIO3, UDMA6, etc.). If `ata_dev_set_mode()` fails, pointer to the failing device is returned in **r_failed_dev**.

LOCKING: PCI/etc. bus probe sem.

Return

0 on success, negative errno otherwise

int **`ata_wait_after_reset`**(struct ata_link * *link*, unsigned long *deadline*, int (*check_ready) (struct ata_link **link*)
wait for link to become ready after reset

Parameters

`struct ata_link * link` link to be waited on

`unsigned long deadline` deadline jiffies for the operation

`int (*)(struct ata_link *link) check_ready` callback to check link readiness

Description

Wait for **link** to become ready after reset.

LOCKING: EH context.

Return

0 if **link** is ready before **deadline**; otherwise, -errno.

int **`sata_link_debounce`**(struct ata_link * *link*, const unsigned long * *params*, unsigned long *deadline*)
debounce SATA phy status

Parameters

`struct ata_link * link` ATA link to debounce SATA phy status for

`const unsigned long * params` timing parameters { interval, duration, timeout } in msec

`unsigned long deadline` deadline jiffies for the operation

Description

Make sure SStatus of **link** reaches stable state, determined by holding the same value where DET is not 1 for **duration** polled every **interval**, before **timeout**. Timeout constraints the beginning of the stable state. Because DET gets stuck at 1 on some controllers after hot unplugging, this functions waits until timeout then returns 0 if DET is stable at 1.

timeout is further limited by **deadline**. The sooner of the two is used.

LOCKING: Kernel thread context (may sleep)

Return

0 on success, -errno on failure.

int **`sata_link_resume`**(struct ata_link * *link*, const unsigned long * *params*, unsigned long *deadline*)
resume SATA link

Parameters

`struct ata_link * link` ATA link to resume SATA

`const unsigned long * params` timing parameters { interval, duration, timeout } in msec

`unsigned long deadline` deadline jiffies for the operation

Description

Resume SATA phy **link** and debounce it.

LOCKING: Kernel thread context (may sleep)

Return

0 on success, -errno on failure.

int **`sata_link_scr_lpm`**(struct ata_link * *link*, enum ata_lpm_policy *policy*, bool *spm_wakeup*)
manipulate SControl IPM and SPM fields

Parameters

`struct ata_link * link` ATA link to manipulate SControl for

`enum ata_lpm_policy policy` LPM policy to configure

`bool spm_wakeup` initiate LPM transition to active state

Description

Manipulate the IPM field of the SControl register of **link** according to **policy**. If **policy** is ATA_LPM_MAX_POWER and **spm_wakeup** is `true`, the SPM field is manipulated to wake up the link. This function also clears PHYRDY_CHG before returning.

LOCKING: EH context.

Return

0 on success, -errno otherwise.

int **`ata_std_prereset`**(struct ata_link * *link*, unsigned long *deadline*)
prepare for reset

Parameters

`struct ata_link * link` ATA link to be reset

`unsigned long deadline` deadline jiffies for the operation

Description

link is about to be reset. Initialize it. Failure from prereset makes libata abort whole reset sequence and give up that port, so prereset should be best-effort. It does its best to prepare for reset sequence but if things go wrong, it should just whine, not fail.

LOCKING: Kernel thread context (may sleep)

Return

0 on success, -errno otherwise.

int **`sata_link_hardreset`**(struct ata_link * *link*, const unsigned long * *timing*, unsigned long *deadline*, bool * *online*, int (*check_ready) (struct ata_link *)
reset link via SATA phy reset

Parameters

`struct ata_link * link` link to reset

`const unsigned long * timing` timing parameters { interval, duration, timeout } in msec

`unsigned long deadline` deadline jiffies for the operation

`bool * online` optional out parameter indicating link onlineness

`int (*)(struct ata_link *) check_ready` optional callback to check link readiness

Description

SATA phy-reset **link** using DET bits of SControl register. After hardreset, link readiness is waited upon using `ata_wait_ready()` if **check_ready** is specified. LLDs are allowed to not specify **check_ready** and wait itself after this function returns. Device classification is LLD's responsibility.

***online** is set to one iff reset succeeded and **link** is online after reset.

LOCKING: Kernel thread context (may sleep)

Return

0 on success, -errno otherwise.

int **sata_std_hardreset**(struct ata_link * *link*, unsigned int * *class*, unsigned long *deadline*)
COMRESET w/o waiting or classification

Parameters

struct ata_link * link link to reset

unsigned int * class resulting class of attached device

unsigned long deadline deadline jiffies for the operation

Description

Standard SATA COMRESET w/o waiting or classification.

LOCKING: Kernel thread context (may sleep)

Return

0 if link offline, -EAGAIN if link online, -errno on errors.

void **ata_std_postreset**(struct ata_link * *link*, unsigned int * *classes*)
standard postreset callback

Parameters

struct ata_link * link the target ata_link

unsigned int * classes classes of attached devices

Description

This function is invoked after a successful reset. Note that the device might have been reset more than once using different reset methods before postreset is invoked.

LOCKING: Kernel thread context (may sleep)

unsigned int **ata_dev_set_feature**(struct ata_device * *dev*, u8 *enable*, u8 *feature*)
Issue SET FEATURES - SATA FEATURES

Parameters

struct ata_device * dev Device to which command will be sent

u8 enable Whether to enable or disable the feature

u8 feature The sector count represents the feature to set

Description

Issue SET FEATURES - SATA FEATURES command to device **dev** on port **ap** with sector count

LOCKING: PCI/etc. bus probe sem.

Return

0 on success, AC_ERR_* mask otherwise.

int **ata_std_qc_defer**(struct ata_queued_cmd * *qc*)
Check whether a qc needs to be deferred

Parameters

struct ata_queued_cmd * qc ATA command in question

Description

Non-NCQ commands cannot run with any other command, NCQ or not. As upper layer only knows the queue depth, we are responsible for maintaining exclusion. This function checks whether a new command **qc** can be issued.

LOCKING: spin_lock_irqsave(host lock)

Return

ATA_DEFER_* if deferring is needed, 0 otherwise.

void **ata_sg_init**(struct ata_queued_cmd * *qc*, struct scatterlist * *sg*, unsigned int *n_elem*)
Associate command with scatter-gather table.

Parameters

struct ata_queued_cmd * qc Command to be associated

struct scatterlist * sg Scatter-gather table.

unsigned int n_elem Number of elements in s/g table.

Description

Initialize the data-related elements of queued_cmd **qc** to point to a scatter-gather table **sg**, containing **n_elem** elements.

LOCKING: spin_lock_irqsave(host lock)

void **ata_qc_complete**(struct ata_queued_cmd * *qc*)
Complete an active ATA command

Parameters

struct ata_queued_cmd * qc Command to complete

Description

Indicate to the mid and upper layers that an ATA command has completed, with either an ok or not-ok status.

Refrain from calling this function multiple times when successfully completing multiple NCQ commands. *ata_qc_complete_multiple()* should be used instead, which will properly update IRQ expect state.

LOCKING: spin_lock_irqsave(host lock)

int **ata_qc_complete_multiple**(struct ata_port * *ap*, u32 *qc_active*)
Complete multiple qcs successfully

Parameters

struct ata_port * ap port in question

u32 qc_active new qc_active mask

Description

Complete in-flight commands. This functions is meant to be called from low-level driver's interrupt routine to complete requests normally. ap->qc_active and **qc_active** is compared and commands are completed accordingly.

Always use this function when completing multiple NCQ commands from IRQ handlers instead of calling *ata_qc_complete()* multiple times to keep IRQ expect status properly in sync.

LOCKING: spin_lock_irqsave(host lock)

Return

Number of completed commands on success, -errno otherwise.

int **sata_scr_valid**(struct ata_link * *link*)
test whether SCRs are accessible

Parameters

struct ata_link * link ATA link to test SCR accessibility for

Description

Test whether SCRs are accessible for **link**.

LOCKING: None.

Return

1 if SCRs are accessible, 0 otherwise.

int **sata_scr_read**(struct ata_link * *link*, int *reg*, u32 * *val*)
read SCR register of the specified port

Parameters

struct ata_link * link ATA link to read SCR for

int reg SCR to read

u32 * val Place to store read value

Description

Read SCR register **reg** of **link** into ***val**. This function is guaranteed to succeed if **link** is ap->link, the cable type of the port is SATA and the port implements ->scr_read.

LOCKING: None if **link** is ap->link. Kernel thread context otherwise.

Return

0 on success, negative errno on failure.

int **sata_scr_write**(struct ata_link * *link*, int *reg*, u32 *val*)
write SCR register of the specified port

Parameters

struct ata_link * link ATA link to write SCR for

int reg SCR to write

u32 val value to write

Description

Write **val** to SCR register **reg** of **link**. This function is guaranteed to succeed if **link** is ap->link, the cable type of the port is SATA and the port implements ->scr_read.

LOCKING: None if **link** is ap->link. Kernel thread context otherwise.

Return

0 on success, negative errno on failure.

int **sata_scr_write_flush**(struct ata_link * *link*, int *reg*, u32 *val*)
write SCR register of the specified port and flush

Parameters

struct ata_link * link ATA link to write SCR for

int reg SCR to write

u32 val value to write

Description

This function is identical to *sata_scr_write()* except that this function performs flush after writing to the register.

LOCKING: None if **link** is ap->link. Kernel thread context otherwise.

Return

0 on success, negative errno on failure.

bool **ata_link_online**(struct ata_link * *link*)
test whether the given link is online

Parameters

struct ata_link * link ATA link to test

Description

Test whether **link** is online. This is identical to *ata_phys_link_online()* when there's no slave link. When there's a slave link, this function should only be called on the master link and will return true if any of M/S links is online.

LOCKING: None.

Return

True if the port online status is available and online.

bool **ata_link_offline**(struct ata_link * *link*)
test whether the given link is offline

Parameters

struct ata_link * link ATA link to test

Description

Test whether **link** is offline. This is identical to *ata_phys_link_offline()* when there's no slave link. When there's a slave link, this function should only be called on the master link and will return true if both M/S links are offline.

LOCKING: None.

Return

True if the port offline status is available and offline.

int **ata_host_suspend**(struct ata_host * *host*, pm_message_t *mesg*)
suspend host

Parameters

struct ata_host * host host to suspend

pm_message_t mesg PM message

Description

Suspend **host**. Actual operation is performed by port suspend.

void **ata_host_resume**(struct ata_host * *host*)
resume host

Parameters

struct ata_host * host host to resume

Description

Resume **host**. Actual operation is performed by port resume.

struct ata_host * **ata_host_alloc**(struct *device* * *dev*, int *max_ports*)
allocate and init basic ATA host resources

Parameters

struct device * dev generic device this host is associated with

int max_ports maximum number of ATA ports associated with this host

Description

Allocate and initialize basic ATA host resources. LLD calls this function to allocate a host, initializes it fully and attaches it using *ata_host_register()*.

max_ports ports are allocated and host->n_ports is initialized to **max_ports**. The caller is allowed to decrease host->n_ports before calling *ata_host_register()*. The unused ports will be automatically freed on registration.

Return

Allocate ATA host on success, NULL on failure.

LOCKING: Inherited from calling layer (may sleep).

struct ata_host * **ata_host_alloc_pinfo**(struct *device* * *dev*, const struct ata_port_info *const * *ppi*, int *n_ports*)
alloc host and init with port_info array

Parameters

struct device * dev generic device this host is associated with

const struct ata_port_info *const * ppi array of ATA port_info to initialize host with

int n_ports number of ATA ports attached to this host

Description

Allocate ATA host and initialize with info from **ppi**. If NULL terminated, **ppi** may contain fewer entries than **n_ports**. The last entry will be used for the remaining ports.

Return

Allocate ATA host on success, NULL on failure.

LOCKING: Inherited from calling layer (may sleep).

int **ata_slave_link_init**(struct ata_port * *ap*)
initialize slave link

Parameters

struct ata_port * ap port to initialize slave link for

Description

Create and initialize slave link for **ap**. This enables slave link handling on the port.

In libata, a port contains links and a link contains devices. There is single host link but if a PMP is attached to it, there can be multiple fan-out links. On SATA, there's usually a single device connected to a link but PATA and SATA controllers emulating TF based interface can have two - master and slave.

However, there are a few controllers which don't fit into this abstraction too well - SATA controllers which emulate TF interface with both master and slave devices but also have separate SCR register sets for each device. These controllers need separate links for physical link handling (e.g. onlineness, link speed) but should be treated like a traditional M/S controller for everything else (e.g. command issue, softreset).

slave_link is libata's way of handling this class of controllers without impacting core layer too much. For anything other than physical link handling, the default host link is used for both master and slave. For physical link handling, separate **ap**->slave_link is used. All dirty details are implemented inside libata core layer. From LLD's POV, the only difference is that prereset, hardreset and postreset are called once more for the slave link, so the reset sequence looks like the following.

prereset(M) -> prereset(S) -> hardreset(M) -> hardreset(S) -> softreset(M) -> postreset(M) -> postreset(S)

Note that softreset is called only for the master. Softreset resets both M/S by definition, so SRST on master should handle both (the standard method will work just fine).

LOCKING: Should be called before host is registered.

Return

0 on success, -errno on failure.

int **`ata_host_start`**(struct ata_host * *host*)
start and freeze ports of an ATA host

Parameters

`struct ata_host * host` ATA host to start ports for

Description

Start and then freeze ports of **host**. Started status is recorded in host->flags, so this function can be called multiple times. Ports are guaranteed to get started only once. If host->ops isn't initialized yet, its set to the first non-dummy port ops.

LOCKING: Inherited from calling layer (may sleep).

Return

0 if all ports are started successfully, -errno otherwise.

void **`ata_host_init`**(struct ata_host * *host*, struct device * *dev*, struct ata_port_operations * *ops*)
Initialize a host struct for sas (ipr, libsas)

Parameters

`struct ata_host * host` host to initialize

`struct device * dev` device host is attached to

`struct ata_port_operations * ops` port_ops

int **`ata_host_register`**(struct ata_host * *host*, struct scsi_host_template * *sht*)
register initialized ATA host

Parameters

`struct ata_host * host` ATA host to register

`struct scsi_host_template * sht` template for SCSI host

Description

Register initialized ATA host. **host** is allocated using *`ata_host_alloc()`* and fully initialized by LLD. This function starts ports, registers **host** with ATA and SCSI layers and probe registered devices.

LOCKING: Inherited from calling layer (may sleep).

Return

0 on success, -errno otherwise.

int **`ata_host_activate`**(struct ata_host * *host*, int *irq*, irq_handler_t *irq_handler*, unsigned long *irq_flags*, struct scsi_host_template * *sht*)
start host, request IRQ and register it

Parameters

`struct ata_host * host` target ATA host

int irq IRQ to request

irq_handler_t irq_handler irq_handler used when requesting IRQ

unsigned long irq_flags irq_flags used when requesting IRQ

struct scsi_host_template * sht scsi_host_template to use when registering the host

Description

After allocating an ATA host and initializing it, most libata LLDs perform three steps to activate the host - start host, request IRQ and register it. This helper takes necessary arguments and performs the three steps in one go.

An invalid IRQ skips the IRQ registration and expects the host to have set polling mode on the port. In this case, **irq_handler** should be NULL.

LOCKING: Inherited from calling layer (may sleep).

Return

0 on success, -errno otherwise.

void **ata_host_detach**(struct ata_host * *host*)
Detach all ports of an ATA host

Parameters

struct ata_host * host Host to detach

Description

Detach all ports of **host**.

LOCKING: Kernel thread context (may sleep).

void **ata_pci_remove_one**(struct pci_dev * *pdev*)
PCI layer callback for device removal

Parameters

struct pci_dev * pdev PCI device that was removed

Description

PCI layer indicates to libata via this hook that hot-unplug or module unload event has occurred. Detach all ports. Resource release is handled via devres.

LOCKING: Inherited from PCI layer (may sleep).

int **ata_platform_remove_one**(struct platform_device * *pdev*)
Platform layer callback for device removal

Parameters

struct platform_device * pdev Platform device that was removed

Description

Platform layer indicates to libata via this hook that hot-unplug or module unload event has occurred. Detach all ports. Resource release is handled via devres.

LOCKING: Inherited from platform layer (may sleep).

void **ata_msleep**(struct ata_port * *ap*, unsigned int *msecs*)
ATA EH owner aware msleep

Parameters

struct ata_port * ap ATA port to attribute the sleep to

unsigned int msecs duration to sleep in milliseconds

Description

Sleeps **msecs**. If the current task is owner of **ap**'s EH, the ownership is released before going to sleep and reacquired after the sleep is complete. IOW, other ports sharing the **ap**->host will be allowed to own the EH while this task is sleeping.

LOCKING: Might sleep.

u32 **ata_wait_register**(struct ata_port * *ap*, void __iomem * *reg*, u32 *mask*, u32 *val*, unsigned long *interval*, unsigned long *timeout*)
wait until register value changes

Parameters

struct ata_port * ap ATA port to wait register for, can be NULL

void __iomem * reg IO-mapped register

u32 mask Mask to apply to read register value

u32 val Wait condition

unsigned long interval polling interval in milliseconds

unsigned long timeout timeout in milliseconds

Description

Waiting for some bits of register to change is a common operation for ATA controllers. This function reads 32bit LE IO-mapped register **reg** and tests for the following condition.

(***reg** & mask) != val

If the condition is met, it returns; otherwise, the process is repeated after **interval_msec** until timeout.

LOCKING: Kernel thread context (may sleep)

Return

The final register value.

bool **sata_lpm_ignore_phy_events**(struct ata_link * *link*)
test if PHY event should be ignored

Parameters

struct ata_link * link Link receiving the event

Description

Test whether the received PHY event has to be ignored or not.

LOCKING: None:

Return

True if the event has to be ignored.

libata Core Internals

struct ata_link * **ata_dev_phys_link**(struct ata_device * *dev*)
find physical link for a device

Parameters

struct ata_device * dev ATA device to look up physical link for

Description

Look up physical link which **dev** is attached to. Note that this is different from **dev**->link only when **dev** is on slave link. For all other cases, it's the same as **dev**->link.

LOCKING: Don't care.

Return

Pointer to the found physical link.

void **`ata_force_cbl`**(struct ata_port * *ap*)
force cable type according to libata.force

Parameters

`struct ata_port * ap` ATA port of interest

Description

Force cable type according to libata.force and whine about it. The last entry which has matching port number is used, so it can be specified as part of device force parameters. For example, both "a:40c,1.00:udma4" and "1.00:40c,udma4" have the same effect.

LOCKING: EH context.

void **`ata_force_link_limits`**(struct ata_link * *link*)
force link limits according to libata.force

Parameters

`struct ata_link * link` ATA link of interest

Description

Force link flags and SATA spd limit according to libata.force and whine about it. When only the port part is specified (e.g. 1:), the limit applies to all links connected to both the host link and all fan-out ports connected via PMP. If the device part is specified as 0 (e.g. 1.00:), it specifies the first fan-out link not the host link. Device number 15 always points to the host link whether PMP is attached or not. If the controller has slave link, device number 16 points to it.

LOCKING: EH context.

void **`ata_force_xfermask`**(struct ata_device * *dev*)
force xfermask according to libata.force

Parameters

`struct ata_device * dev` ATA device of interest

Description

Force xfer_mask according to libata.force and whine about it. For consistency with link selection, device number 15 selects the first device connected to the host link.

LOCKING: EH context.

void **`ata_force_horkage`**(struct ata_device * *dev*)
force horkage according to libata.force

Parameters

`struct ata_device * dev` ATA device of interest

Description

Force horkage according to libata.force and whine about it. For consistency with link selection, device number 15 selects the first device connected to the host link.

LOCKING: EH context.

int **`ata_rwcmd_protocol`**(struct ata_taskfile * *tf*, struct ata_device * *dev*)
set taskfile r/w commands and protocol

Parameters

`struct ata_taskfile * tf` command to examine and configure

`struct ata_device * dev` device tf belongs to

Description

Examine the device configuration and tf->flags to calculate the proper read/write commands and protocol to use.

LOCKING: caller.

u64 **`ata_tf_read_block`**(const struct ata_taskfile * *tf*, struct ata_device * *dev*)
Read block address from ATA taskfile

Parameters

`const struct ata_taskfile * tf` ATA taskfile of interest

`struct ata_device * dev` ATA device **tf** belongs to

Description

LOCKING: None.

Read block address from **tf**. This function can handle all three address formats - LBA, LBA48 and CHS. tf->protocol and flags select the address format to use.

Return

Block address read from **tf**.

int **`ata_build_rw_tf`**(struct ata_taskfile * *tf*, struct ata_device * *dev*, u64 *block*, u32 *n_block*, unsigned int *tf_flags*, unsigned int *tag*, int *class*)
Build ATA taskfile for given read/write request

Parameters

`struct ata_taskfile * tf` Target ATA taskfile

`struct ata_device * dev` ATA device **tf** belongs to

`u64 block` Block address

`u32 n_block` Number of blocks

`unsigned int tf_flags` RW/FUA etc...

`unsigned int tag` tag

`int class` IO priority class

Description

LOCKING: None.

Build ATA taskfile **tf** for read/write request described by **block**, **n_block**, **tf_flags** and **tag** on **dev**.

Return

0 on success, -ERANGE if the request is too large for **dev**, -EINVAL if the request is invalid.

int **`ata_read_native_max_address`**(struct ata_device * *dev*, u64 * *max_sectors*)
Read native max address

Parameters

`struct ata_device * dev` target device

`u64 * max_sectors` out parameter for the result native max address

Description

Perform an LBA48 or LBA28 native size query upon the device in question.

Return

0 on success, -EACCES if command is aborted by the drive. -EIO on other errors.

int **ata_set_max_sectors**(struct ata_device * *dev*, u64 *new_sectors*)
Set max sectors

Parameters

struct ata_device * dev target device

u64 new_sectors new max sectors value to set for the device

Description

Set max sectors of **dev** to **new_sectors**.

Return

0 on success, -EACCES if command is aborted or denied (due to previous non-volatile SET_MAX) by the drive. -EIO on other errors.

int **ata_hpa_resize**(struct ata_device * *dev*)
Resize a device with an HPA set

Parameters

struct ata_device * dev Device to resize

Description

Read the size of an LBA28 or LBA48 disk with HPA features and resize it if required to the full size of the media. The caller must check the drive has the HPA feature set enabled.

Return

0 on success, -errno on failure.

void **ata_dump_id**(const u16 * *id*)
IDENTIFY DEVICE info debugging output

Parameters

const u16 * id IDENTIFY DEVICE page to dump

Description

Dump selected 16-bit words from the given IDENTIFY DEVICE page.

LOCKING: caller.

unsigned **ata_exec_internal_sg**(struct ata_device * *dev*, struct ata_taskfile * *tf*, const u8 * *cdb*, int *dma_dir*, struct scatterlist * *sgl*, unsigned int *n_elem*, unsigned long *timeout*)
execute libata internal command

Parameters

struct ata_device * dev Device to which the command is sent

struct ata_taskfile * tf Taskfile registers for the command and the result

const u8 * cdb CDB for packet command

int dma_dir Data transfer direction of the command

struct scatterlist * sgl sg list for the data buffer of the command

unsigned int n_elem Number of sg entries

unsigned long timeout Timeout in msecs (0 for default)

Description

Executes libata internal command with timeout. **tf** contains command on entry and result on return. Timeout and error conditions are reported via return value. No recovery action is taken after a command times out. It's caller's duty to clean up after timeout.

LOCKING: None. Should be called with kernel context, might sleep.

Return

Zero on success, AC_ERR_* mask on failure

unsigned **ata_exec_internal**(struct ata_device * *dev*, struct ata_taskfile * *tf*, const u8 * *cdb*, int *dma_dir*, void * *buf*, unsigned int *buflen*, unsigned long *timeout*)
execute libata internal command

Parameters

struct ata_device * dev Device to which the command is sent

struct ata_taskfile * tf Taskfile registers for the command and the result

const u8 * cdb CDB for packet command

int dma_dir Data transfer direction of the command

void * buf Data buffer of the command

unsigned int buflen Length of data buffer

unsigned long timeout Timeout in msecs (0 for default)

Description

Wrapper around *ata_exec_internal_sg()* which takes simple buffer instead of sg list.

LOCKING: None. Should be called with kernel context, might sleep.

Return

Zero on success, AC_ERR_* mask on failure

u32 **ata_pio_mask_no_iordy**(const struct ata_device * *adev*)
Return the non IORDY mask

Parameters

const struct ata_device * adev ATA device

Description

Compute the highest mode possible if we are not using iordy. Return -1 if no iordy mode is available.

int **ata_dev_read_id**(struct ata_device * *dev*, unsigned int * *p_class*, unsigned int *flags*, u16 * *id*)
Read ID data from the specified device

Parameters

struct ata_device * dev target device

unsigned int * p_class pointer to class of the target device (may be changed)

unsigned int flags ATA_READID_* flags

u16 * id buffer to read IDENTIFY data into

Description

Read ID data from the specified device. ATA_CMD_ID_ATA is performed on ATA devices and ATA_CMD_ID_ATAPI on ATAPI devices. This function also issues ATA_CMD_INIT_DEV_PARAMS for pre-ATA4 drives.

FIXME: ATA_CMD_ID_ATA is optional for early drives and right now we abort if we hit that case.

LOCKING: Kernel thread context (may sleep)

Return

0 on success, -errno otherwise.

unsigned int **ata_read_log_page**(struct ata_device * *dev*, u8 *log*, u8 *page*, void * *buf*, unsigned int *sectors*)

read a specific log page

Parameters

struct ata_device * dev target device

u8 log log to read

u8 page page to read

void * buf buffer to store read page

unsigned int sectors number of sectors to read

Description

Read log page using READ_LOG_EXT command.

LOCKING: Kernel thread context (may sleep).

Return

0 on success, AC_ERR_* mask otherwise.

int **ata_dev_configure**(struct ata_device * *dev*)

Configure the specified ATA/ATAPI device

Parameters

struct ata_device * dev Target device to configure

Description

Configure **dev** according to **dev**->id. Generic and low-level driver specific fixups are also applied.

LOCKING: Kernel thread context (may sleep)

Return

0 on success, -errno otherwise

int **ata_bus_probe**(struct ata_port * *ap*)

Reset and probe ATA bus

Parameters

struct ata_port * ap Bus to probe

Description

Master ATA bus probing function. Initiates a hardware-dependent bus reset, then attempts to identify any devices found on the bus.

LOCKING: PCI/etc. bus probe sem.

Return

Zero on success, negative errno otherwise.

void **sata_print_link_status**(struct ata_link * *link*)

Print SATA link status

Parameters

struct ata_link * link SATA link to printk link status about

Description

This function prints link speed and status of a SATA link.

LOCKING: None.

int **sata_down_spd_limit**(struct ata_link * *link*, u32 *spd_limit*)
adjust SATA spd limit downward

Parameters

struct ata_link * link Link to adjust SATA spd limit for

u32 spd_limit Additional limit

Description

Adjust SATA spd limit of **link** downward. Note that this function only adjusts the limit. The change must be applied using *sata_set_spd()*.

If **spd_limit** is non-zero, the speed is limited to equal to or lower than **spd_limit** if such speed is supported. If **spd_limit** is slower than any supported speed, only the lowest supported speed is allowed.

LOCKING: Inherited from caller.

Return

0 on success, negative errno on failure

int **sata_set_spd_needed**(struct ata_link * *link*)
is SATA spd configuration needed

Parameters

struct ata_link * link Link in question

Description

Test whether the spd limit in SControl matches **link**->sata_spd_limit. This function is used to determine whether hardreset is necessary to apply SATA spd configuration.

LOCKING: Inherited from caller.

Return

1 if SATA spd configuration is needed, 0 otherwise.

int **ata_down_xfermask_limit**(struct ata_device * *dev*, unsigned int *sel*)
adjust dev xfer masks downward

Parameters

struct ata_device * dev Device to adjust xfer masks

unsigned int sel ATA_DNXFER_* selector

Description

Adjust xfer masks of **dev** downward. Note that this function does not apply the change. Invoking *ata_set_mode()* afterwards will apply the limit.

LOCKING: Inherited from caller.

Return

0 on success, negative errno on failure

int **ata_wait_ready**(struct ata_link * *link*, unsigned long *deadline*, int (*check_ready) (struct ata_link **link*)
wait for link to become ready

Parameters

struct ata_link * link link to be waited on

`unsigned long deadline` deadline jiffies for the operation

`int (*)(struct ata_link *link) check_ready` callback to check link readiness

Description

Wait for **link** to become ready. **check_ready** should return positive number if **link** is ready, 0 if it isn't, -ENODEV if link doesn't seem to be occupied, other errno for other error conditions.

Transient -ENODEV conditions are allowed for ATA_TMOUT_FF_WAIT.

LOCKING: EH context.

Return

0 if **link** is ready before **deadline**; otherwise, -errno.

int **`ata_dev_same_device`**(struct ata_device * *dev*, unsigned int *new_class*, const u16 * *new_id*)
Determine whether new ID matches configured device

Parameters

`struct ata_device * dev` device to compare against

`unsigned int new_class` class of the new device

`const u16 * new_id` IDENTIFY page of the new device

Description

Compare **new_class** and **new_id** against **dev** and determine whether **dev** is the device indicated by **new_class** and **new_id**.

LOCKING: None.

Return

1 if **dev** matches **new_class** and **new_id**, 0 otherwise.

int **`ata_dev_reread_id`**(struct ata_device * *dev*, unsigned int *readid_flags*)
Re-read IDENTIFY data

Parameters

`struct ata_device * dev` target ATA device

`unsigned int readid_flags` read ID flags

Description

Re-read IDENTIFY page and make sure **dev** is still attached to the port.

LOCKING: Kernel thread context (may sleep)

Return

0 on success, negative errno otherwise

int **`ata_dev_revalidate`**(struct ata_device * *dev*, unsigned int *new_class*, unsigned int *readid_flags*)
Revalidate ATA device

Parameters

`struct ata_device * dev` device to revalidate

`unsigned int new_class` new class code

`unsigned int readid_flags` read ID flags

Description

Re-read IDENTIFY page, make sure **dev** is still attached to the port and reconfigure it according to the new IDENTIFY page.

LOCKING: Kernel thread context (may sleep)

Return

0 on success, negative errno otherwise

int **`ata_is_40wire`**(struct ata_device * *dev*)
check drive side detection

Parameters

`struct ata_device * dev` device

Description

Perform drive side detection decoding, allowing for device vendors who can't follow the documentation.

int **`cable_is_40wire`**(struct ata_port * *ap*)
40/80/SATA decider

Parameters

`struct ata_port * ap` port to consider

Description

This function encapsulates the policy for speed management in one place. At the moment we don't cache the result but there is a good case for setting ap->cbl to the result when we are called with unknown cables (and figuring out if it impacts hotplug at all).

Return 1 if the cable appears to be 40 wire.

void **`ata_dev_xfermask`**(struct ata_device * *dev*)
Compute supported xfermask of the given device

Parameters

`struct ata_device * dev` Device to compute xfermask for

Description

Compute supported xfermask of **dev** and store it in dev->*_mask. This function is responsible for applying all known limits including host controller limits, device blacklist, etc...

LOCKING: None.

unsigned int **`ata_dev_set_xfermode`**(struct ata_device * *dev*)
Issue SET FEATURES - XFER MODE command

Parameters

`struct ata_device * dev` Device to which command will be sent

Description

Issue SET FEATURES - XFER MODE command to device **dev** on port **ap**.

LOCKING: PCI/etc. bus probe sem.

Return

0 on success, AC_ERR_* mask otherwise.

unsigned int **`ata_dev_init_params`**(struct ata_device * *dev*, u16 *heads*, u16 *sectors*)
Issue INIT DEV PARAMS command

Parameters

`struct ata_device * dev` Device to which command will be sent

u16 heads Number of heads (taskfile parameter)

u16 sectors Number of sectors (taskfile parameter)

Description

LOCKING: Kernel thread context (may sleep)

Return

0 on success, AC_ERR_* mask otherwise.

int **atapi_check_dma**(struct ata_queued_cmd * *qc*)
Check whether ATAPI DMA can be supported

Parameters

struct ata_queued_cmd * qc Metadata associated with taskfile to check

Description

Allow low-level driver to filter ATA PACKET commands, returning a status indicating whether or not it is OK to use DMA for the supplied PACKET command.

LOCKING: spin_lock_irqsave(host lock)

Return

0 when ATAPI DMA can be used nonzero otherwise

void **ata_sg_clean**(struct ata_queued_cmd * *qc*)
Unmap DMA memory associated with command

Parameters

struct ata_queued_cmd * qc Command containing DMA memory to be released

Description

Unmap all mapped DMA memory associated with this command.

LOCKING: spin_lock_irqsave(host lock)

int **ata_sg_setup**(struct ata_queued_cmd * *qc*)
DMA-map the scatter-gather table associated with a command.

Parameters

struct ata_queued_cmd * qc Command with scatter-gather table to be mapped.

Description

DMA-map the scatter-gather table associated with queued_cmd **qc**.

LOCKING: spin_lock_irqsave(host lock)

Return

Zero on success, negative on error.

void **swap_buf_le16**(u16 * *buf*, unsigned int *buf_words*)
swap halves of 16-bit words in place

Parameters

u16 * buf Buffer to swap

unsigned int buf_words Number of 16-bit words in buffer.

Description

Swap halves of 16-bit words if needed to convert from little-endian byte order to native cpu byte order, or vice-versa.

LOCKING: Inherited from caller.

struct ata_queued_cmd * **ata_qc_new_init**(struct ata_device * *dev*, int *tag*)
Request an available ATA command, and initialize it

Parameters

struct ata_device * dev Device from whom we request an available command structure

int tag tag

Description

LOCKING: None.

void **ata_qc_free**(struct ata_queued_cmd * *qc*)
free unused ata_queued_cmd

Parameters

struct ata_queued_cmd * qc Command to complete

Description

Designed to free unused ata_queued_cmd object in case something prevents using it.

LOCKING: spin_lock_irqsave(host lock)

void **ata_qc_issue**(struct ata_queued_cmd * *qc*)
issue taskfile to device

Parameters

struct ata_queued_cmd * qc command to issue to device

Description

Prepare an ATA command to submission to device. This includes mapping the data into a DMA-able area, filling in the S/G table, and finally writing the taskfile to hardware, starting the command.

LOCKING: spin_lock_irqsave(host lock)

bool **ata_phys_link_online**(struct ata_link * *link*)
test whether the given link is online

Parameters

struct ata_link * link ATA link to test

Description

Test whether **link** is online. Note that this function returns 0 if online status of **link** cannot be obtained, so ata_link_online(link) != !ata_link_offline(link).

LOCKING: None.

Return

True if the port online status is available and online.

bool **ata_phys_link_offline**(struct ata_link * *link*)
test whether the given link is offline

Parameters

struct ata_link * link ATA link to test

Description

Test whether **link** is offline. Note that this function returns 0 if offline status of **link** cannot be obtained, so ata_link_online(link) != !ata_link_offline(link).

LOCKING: None.

Return

True if the port offline status is available and offline.

void **`ata_dev_init`**(struct ata_device * *dev*)
Initialize an ata_device structure

Parameters

`struct ata_device * dev` Device structure to initialize

Description

Initialize **dev** in preparation for probing.

LOCKING: Inherited from caller.

void **`ata_link_init`**(struct ata_port * *ap*, struct ata_link * *link*, int *pmp*)
Initialize an ata_link structure

Parameters

`struct ata_port * ap` ATA port link is attached to

`struct ata_link * link` Link structure to initialize

`int pmp` Port multiplier port number

Description

Initialize **link**.

LOCKING: Kernel thread context (may sleep)

int **`sata_link_init_spd`**(struct ata_link * *link*)
Initialize link->sata_spd_limit

Parameters

`struct ata_link * link` Link to configure sata_spd_limit for

Description

Initialize **link**->[**hw_**]sata_spd_limit to the currently configured value.

LOCKING: Kernel thread context (may sleep).

Return

0 on success, -errno on failure.

struct ata_port * **`ata_port_alloc`**(struct ata_host * *host*)
allocate and initialize basic ATA port resources

Parameters

`struct ata_host * host` ATA host this allocated port belongs to

Description

Allocate and initialize basic ATA port resources.

Return

Allocate ATA port on success, NULL on failure.

LOCKING: Inherited from calling layer (may sleep).

void **`ata_finalize_port_ops`**(struct ata_port_operations * *ops*)
finalize ata_port_operations

Parameters

`struct ata_port_operations * ops` ata_port_operations to finalize

Description

An ata_port_operations can inherit from another ops and that ops can again inherit from another. This can go on as many times as necessary as long as there is no loop in the inheritance chain.

Ops tables are finalized when the host is started. NULL or unspecified entries are inherited from the closet ancestor which has the method and the entry is populated with it. After finalization, the ops table directly points to all the methods and ->inherits is no longer necessary and cleared.

Using ATA_OP_NULL, inheriting ops can force a method to NULL.

LOCKING: None.

void **ata_port_detach**(struct ata_port * *ap*)
Detach ATA port in preparation of device removal

Parameters

struct ata_port * ap ATA port to be detached

Description

Detach all ATA devices and the associated SCSI devices of **ap**; then, remove the associated SCSI host. **ap** is guaranteed to be quiescent on return from this function.

LOCKING: Kernel thread context (may sleep).

void **__ata_ehi_push_desc**(struct ata_eh_info * *ehi*, const char * *fmt*, ...)
push error description without adding separator

Parameters

struct ata_eh_info * ehi target EHI

const char * fmt printf format string

... variable arguments

Description

Format string according to **fmt** and append it to **ehi**->desc.

LOCKING: spin_lock_irqsave(host lock)

void **ata_ehi_push_desc**(struct ata_eh_info * *ehi*, const char * *fmt*, ...)
push error description with separator

Parameters

struct ata_eh_info * ehi target EHI

const char * fmt printf format string

... variable arguments

Description

Format string according to **fmt** and append it to **ehi**->desc. If **ehi**->desc is not empty, ", " is added in-between.

LOCKING: spin_lock_irqsave(host lock)

void **ata_ehi_clear_desc**(struct ata_eh_info * *ehi*)
clean error description

Parameters

struct ata_eh_info * ehi target EHI

Description

Clear **ehi**->desc.

LOCKING: spin_lock_irqsave(host lock)

void **ata_port_desc**(struct ata_port * *ap*, const char * *fmt*, ...)
append port description

Parameters

struct ata_port * ap target ATA port

const char * fmt printf format string

... variable arguments

Description

Format string according to **fmt** and append it to port description. If port description is not empty, " " is added in-between. This function is to be used while initializing ata_host. The description is printed on host registration.

LOCKING: None.

void **ata_port_pbar_desc**(struct ata_port * *ap*, int *bar*, ssize_t *offset*, const char * *name*)
append PCI BAR description

Parameters

struct ata_port * ap target ATA port

int bar target PCI BAR

ssize_t offset offset into PCI BAR

const char * name name of the area

Description

If **offset** is negative, this function formats a string which contains the name, address, size and type of the BAR and appends it to the port description. If **offset** is zero or positive, only name and offsetted address is appended.

LOCKING: None.

unsigned long **ata_internal_cmd_timeout**(struct ata_device * *dev*, u8 *cmd*)
determine timeout for an internal command

Parameters

struct ata_device * dev target device

u8 cmd internal command to be issued

Description

Determine timeout for internal command **cmd** for **dev**.

LOCKING: EH context.

Return

Determined timeout.

void **ata_internal_cmd_timed_out**(struct ata_device * *dev*, u8 *cmd*)
notification for internal command timeout

Parameters

struct ata_device * dev target device

u8 cmd internal command which timed out

Description

Notify EH that internal command **cmd** for **dev** timed out. This function should be called only for commands whose timeouts are determined using *ata_internal_cmd_timeout()*.

LOCKING: EH context.

void **ata_eh_acquire**(struct ata_port * *ap*)
acquire EH ownership

Parameters

struct ata_port * ap ATA port to acquire EH ownership for

Description

Acquire EH ownership for **ap**. This is the basic exclusion mechanism for ports sharing a host. Only one port hanging off the same host can claim the ownership of EH.

LOCKING: EH context.

void **ata_eh_release**(struct ata_port * *ap*)
release EH ownership

Parameters

struct ata_port * ap ATA port to release EH ownership for

Description

Release EH ownership for **ap** if the caller. The caller must have acquired EH ownership using *ata_eh_acquire()* previously.

LOCKING: EH context.

enum blk_eh_timer_return **ata_scsi_timed_out**(struct scsi_cmnd * *cmd*)
SCSI layer time out callback

Parameters

struct scsi_cmnd * cmd timed out SCSI command

Description

Handles SCSI layer timeout. We race with normal completion of the qc for **cmd**. If the qc is already gone, we lose and let the scsi command finish (EH_HANDLED). Otherwise, the qc has timed out and EH should be invoked. Prevent *ata_qc_complete()* from finishing it by setting EH_SCHEDULED and return EH_NOT_HANDLED.

TODO: kill this function once old EH is gone.

LOCKING: Called from timer context

Return

EH_HANDLED or EH_NOT_HANDLED

void **ata_scsi_error**(struct Scsi_Host * *host*)
SCSI layer error handler callback

Parameters

struct Scsi_Host * host SCSI host on which error occurred

Description

Handles SCSI-layer-thrown error events.

LOCKING: Inherited from SCSI layer (none, can sleep)

Return

Zero.

void **ata_scsi_cmd_error_handler**(struct Scsi_Host * *host*, struct ata_port * *ap*, struct list_head * *eh_work_q*)
error callback for a list of commands

Parameters

struct Scsi_Host * host scsi host containing the port

struct ata_port * ap ATA port within the host

struct list_head * eh_work_q list of commands to process

Description

process the given list of commands and return those finished to the ap->eh_done_q. This function is the first part of the libata error handler which processes a given list of failed commands.

void **ata_scsi_port_error_handler**(struct Scsi_Host * *host*, struct ata_port * *ap*)
recover the port after the commands

Parameters

struct Scsi_Host * host SCSI host containing the port

struct ata_port * ap the ATA port

Description

Handle the recovery of the port **ap** after all the commands have been recovered.

void **ata_port_wait_eh**(struct ata_port * *ap*)
Wait for the currently pending EH to complete

Parameters

struct ata_port * ap Port to wait EH for

Description

Wait until the currently pending EH is complete.

LOCKING: Kernel thread context (may sleep).

void **ata_eh_set_pending**(struct ata_port * *ap*, int *fastdrain*)
set ATA_PFLAG_EH_PENDING and activate fast drain

Parameters

struct ata_port * ap target ATA port

int fastdrain activate fast drain

Description

Set ATA_PFLAG_EH_PENDING and activate fast drain if **fastdrain** is non-zero and EH wasn't pending before. Fast drain ensures that EH kicks in in timely manner.

LOCKING: spin_lock_irqsave(host lock)

void **ata_qc_schedule_eh**(struct ata_queued_cmd * *qc*)
schedule qc for error handling

Parameters

struct ata_queued_cmd * qc command to schedule error handling for

Description

Schedule error handling for **qc**. EH will kick in as soon as other commands are drained.

LOCKING: spin_lock_irqsave(host lock)

void **ata_std_sched_eh**(struct ata_port * *ap*)
non-libsas ata_ports issue eh with this common routine

Parameters

struct ata_port * ap ATA port to schedule EH for

Description

LOCKING: inherited from ata_port_schedule_eh spin_lock_irqsave(host lock)

void **ata_std_end_eh**(struct ata_port * *ap*)
non-libsas ata_ports complete eh with this common routine

Parameters

struct ata_port * ap ATA port to end EH for

Description

In the libata object model there is a 1:1 mapping of ata_port to shost, so host fields can be directly manipulated under ap->lock, in the libsas case we need to hold a lock at the ha->level to coordinate these events.

LOCKING: spin_lock_irqsave(host lock)

void **ata_port_schedule_eh**(struct ata_port * *ap*)
schedule error handling without a qc

Parameters

struct ata_port * ap ATA port to schedule EH for

Description

Schedule error handling for **ap**. EH will kick in as soon as all commands are drained.

LOCKING: spin_lock_irqsave(host lock)

int **ata_link_abort**(struct ata_link * *link*)
abort all qc's on the link

Parameters

struct ata_link * link ATA link to abort qc's for

Description

Abort all active qc's active on **link** and schedule EH.

LOCKING: spin_lock_irqsave(host lock)

Return

Number of aborted qc's.

int **ata_port_abort**(struct ata_port * *ap*)
abort all qc's on the port

Parameters

struct ata_port * ap ATA port to abort qc's for

Description

Abort all active qc's of **ap** and schedule EH.

LOCKING: spin_lock_irqsave(host_set lock)

Return

Number of aborted qc's.

void **__ata_port_freeze**(struct ata_port * *ap*)
freeze port

Parameters

struct ata_port * ap ATA port to freeze

Description

This function is called when HSM violation or some other condition disrupts normal operation of the port. Frozen port is not allowed to perform any operation until the port is thawed, which usually follows a successful reset.

ap->ops->:c:func:*freeze()* callback can be used for freezing the port hardware-wise (e.g. mask interrupt and stop DMA engine). If a port cannot be frozen hardware-wise, the interrupt handler must ack and clear interrupts unconditionally while the port is frozen.

LOCKING: spin_lock_irqsave(host lock)

int **ata_port_freeze**(struct ata_port * *ap*)
abort & freeze port

Parameters

struct ata_port * ap ATA port to freeze

Description

Abort and freeze **ap**. The freeze operation must be called first, because some hardware requires special operations before the taskfile registers are accessible.

LOCKING: spin_lock_irqsave(host lock)

Return

Number of aborted commands.

int **sata_async_notification**(struct ata_port * *ap*)
SATA async notification handler

Parameters

struct ata_port * ap ATA port where async notification is received

Description

Handler to be called when async notification via SDB FIS is received. This function schedules EH if necessary.

LOCKING: spin_lock_irqsave(host lock)

Return

1 if EH is scheduled, 0 otherwise.

void **ata_eh_freeze_port**(struct ata_port * *ap*)
EH helper to freeze port

Parameters

struct ata_port * ap ATA port to freeze

Description

Freeze **ap**.

LOCKING: None.

void **ata_eh_thaw_port**(struct ata_port * *ap*)
EH helper to thaw port

Parameters

struct ata_port * ap ATA port to thaw

Description

Thaw frozen port **ap**.

LOCKING: None.

void **ata_eh_qc_complete**(struct ata_queued_cmd * *qc*)
Complete an active ATA command from EH

Parameters

`struct ata_queued_cmd * qc` Command to complete

Description

Indicate to the mid and upper layers that an ATA command has completed. To be used from EH.

void **`ata_eh_qc_retry`**(struct ata_queued_cmd * *qc*)
Tell midlayer to retry an ATA command after EH

Parameters

`struct ata_queued_cmd * qc` Command to retry

Description

Indicate to the mid and upper layers that an ATA command should be retried. To be used from EH.

SCSI midlayer limits the number of retries to scmd->allowed. scmd->allowed is incremented for commands which get retried due to unrelated failures (qc->err_mask is zero).

void **`ata_dev_disable`**(struct ata_device * *dev*)
disable ATA device

Parameters

`struct ata_device * dev` ATA device to disable

Description

Disable **dev**.

Locking: EH context.

void **`ata_eh_detach_dev`**(struct ata_device * *dev*)
detach ATA device

Parameters

`struct ata_device * dev` ATA device to detach

Description

Detach **dev**.

LOCKING: None.

void **`ata_eh_about_to_do`**(struct ata_link * *link*, struct ata_device * *dev*, unsigned int *action*)
about to perform eh_action

Parameters

`struct ata_link * link` target ATA link

`struct ata_device * dev` target ATA dev for per-dev action (can be NULL)

`unsigned int action` action about to be performed

Description

Called just before performing EH actions to clear related bits in **link**->eh_info such that eh actions are not unnecessarily repeated.

LOCKING: None.

void **`ata_eh_done`**(struct ata_link * *link*, struct ata_device * *dev*, unsigned int *action*)
EH action complete

Parameters

`struct ata_link * link` ATA link for which EH actions are complete

`struct ata_device * dev` target ATA dev for per-dev action (can be NULL)

unsigned int action action just completed

Description

Called right after performing EH actions to clear related bits in **link**->eh_context.

LOCKING: None.

const char * **ata_err_string**(unsigned int *err_mask*)
convert err_mask to descriptive string

Parameters

unsigned int err_mask error mask to convert to string

Description

Convert **err_mask** to descriptive string. Errors are prioritized according to severity and only the most severe error is reported.

LOCKING: None.

Return

Descriptive string for **err_mask**

int **ata_eh_read_log_10h**(struct ata_device * *dev*, int * *tag*, struct ata_taskfile * *tf*)
Read log page 10h for NCQ error details

Parameters

struct ata_device * dev Device to read log page 10h from

int * tag Resulting tag of the failed command

struct ata_taskfile * tf Resulting taskfile registers of the failed command

Description

Read log page 10h to obtain NCQ error details and clear error condition.

LOCKING: Kernel thread context (may sleep).

Return

0 on success, -errno otherwise.

unsigned int **atapi_eh_tur**(struct ata_device * *dev*, u8 * *r_sense_key*)
perform ATAPI TEST_UNIT_READY

Parameters

struct ata_device * dev target ATAPI device

u8 * r_sense_key out parameter for sense_key

Description

Perform ATAPI TEST_UNIT_READY.

LOCKING: EH context (may sleep).

Return

0 on success, AC_ERR_* mask on failure.

void **ata_eh_request_sense**(struct ata_queued_cmd * *qc*, struct scsi_cmnd * *cmd*)
perform REQUEST_SENSE_DATA_EXT

Parameters

struct ata_queued_cmd * qc qc to perform REQUEST_SENSE_SENSE_DATA_EXT to

struct scsi_cmnd * cmd scsi command for which the sense code should be set

Description

Perform REQUEST_SENSE_DATA_EXT after the device reported CHECK SENSE. This function is an EH helper.

LOCKING: Kernel thread context (may sleep).

unsigned int **atapi_eh_request_sense**(struct ata_device * *dev*, u8 * *sense_buf*, u8 *dfl_sense_key*)
perform ATAPI REQUEST_SENSE

Parameters

struct ata_device * dev device to perform REQUEST_SENSE to

u8 * sense_buf result sense data buffer (SCSI_SENSE_BUFFERSIZE bytes long)

u8 dfl_sense_key default sense key to use

Description

Perform ATAPI REQUEST_SENSE after the device reported CHECK SENSE. This function is EH helper.

LOCKING: Kernel thread context (may sleep).

Return

0 on success, AC_ERR_* mask on failure

void **ata_eh_analyze_serror**(struct ata_link * *link*)
analyze SError for a failed port

Parameters

struct ata_link * link ATA link to analyze SError for

Description

Analyze SError if available and further determine cause of failure.

LOCKING: None.

void **ata_eh_analyze_ncq_error**(struct ata_link * *link*)
analyze NCQ error

Parameters

struct ata_link * link ATA link to analyze NCQ error for

Description

Read log page 10h, determine the offending qc and acquire error status TF. For NCQ device errors, all LLDDs have to do is setting AC_ERR_DEV in ehi->err_mask. This function takes care of the rest.

LOCKING: Kernel thread context (may sleep).

unsigned int **ata_eh_analyze_tf**(struct ata_queued_cmd * *qc*, const struct ata_taskfile * *tf*)
analyze taskfile of a failed qc

Parameters

struct ata_queued_cmd * qc qc to analyze

const struct ata_taskfile * tf Taskfile registers to analyze

Description

Analyze taskfile of **qc** and further determine cause of failure. This function also requests ATAPI sense data if available.

LOCKING: Kernel thread context (may sleep).

Return

Determined recovery action

unsigned int **`ata_eh_speed_down_verdict`**(struct ata_device * *dev*)
Determine speed down verdict

Parameters

`struct ata_device * dev` Device of interest

Description

This function examines error ring of **dev** and determines whether NCQ needs to be turned off, transfer speed should be stepped down, or falling back to PIO is necessary.

ECAT_ATA_BUS : ATA_BUS error for any command

ECAT_TOUT_HSM [TIMEOUT for any command or HSM violation for] IO commands

ECAT_UNK_DEV : Unknown DEV error for IO commands

ECAT_DUBIOUS_* [Identical to above three but occurred while] data transfer hasn't been verified.

Verdicts are

NCQ_OFF : Turn off NCQ.

SPEED_DOWN [Speed down transfer speed but don't fall back] to PIO.

FALLBACK_TO_PIO : Fall back to PIO.

Even if multiple verdicts are returned, only one action is taken per error. An action triggered by non-DUBIOUS errors clears ering, while one triggered by DUBIOUS_* errors doesn't. This is to expedite speed down decisions right after device is initially configured.

The following are speed down rules. #1 and #2 deal with DUBIOUS errors.

1. If more than one DUBIOUS_ATA_BUS or DUBIOUS_TOUT_HSM errors occurred during last 5 mins, SPEED_DOWN and FALLBACK_TO_PIO.
2. If more than one DUBIOUS_TOUT_HSM or DUBIOUS_UNK_DEV errors occurred during last 5 mins, NCQ_OFF.
3. If more than 8 ATA_BUS, TOUT_HSM or UNK_DEV errors occurred during last 5 mins, FALLBACK_TO_PIO
4. If more than 3 TOUT_HSM or UNK_DEV errors occurred during last 10 mins, NCQ_OFF.
5. If more than 3 ATA_BUS or TOUT_HSM errors, or more than 6 UNK_DEV errors occurred during last 10 mins, SPEED_DOWN.

LOCKING: Inherited from caller.

Return

OR of ATA_EH_SPDN_* flags.

unsigned int **`ata_eh_speed_down`**(struct ata_device * *dev*, unsigned int *eflags*, unsigned int *err_mask*)
record error and speed down if necessary

Parameters

`struct ata_device * dev` Failed device

`unsigned int eflags` mask of ATA_EFLAG_* flags

`unsigned int err_mask` err_mask of the error

Description

Record error and examine error history to determine whether adjusting transmission speed is necessary. It also sets transmission limits appropriately if such adjustment is necessary.

LOCKING: Kernel thread context (may sleep).

Return

Determined recovery action.

int **ata_eh_worth_retry**(struct ata_queued_cmd * *qc*)
analyze error and decide whether to retry

Parameters

struct ata_queued_cmd * qc qc to possibly retry

Description

Look at the cause of the error and decide if a retry might be useful or not. We don't want to retry media errors because the drive itself has probably already taken 10-30 seconds doing its own internal retries before reporting the failure.

void **ata_eh_link_autopsy**(struct ata_link * *link*)
analyze error and determine recovery action

Parameters

struct ata_link * link host link to perform autopsy on

Description

Analyze why **link** failed and determine which recovery actions are needed. This function also sets more detailed AC_ERR_* values and fills sense data for ATAPI CHECK SENSE.

LOCKING: Kernel thread context (may sleep).

void **ata_eh_autopsy**(struct ata_port * *ap*)
analyze error and determine recovery action

Parameters

struct ata_port * ap host port to perform autopsy on

Description

Analyze all links of **ap** and determine why they failed and which recovery actions are needed.

LOCKING: Kernel thread context (may sleep).

const char * **ata_get_cmd_descript**(u8 *command*)
get description for ATA command

Parameters

u8 command ATA command code to get description for

Description

Return a textual description of the given command, or NULL if the command is not known.

LOCKING: None

void **ata_eh_link_report**(struct ata_link * *link*)
report error handling to user

Parameters

struct ata_link * link ATA link EH is going on

Description

Report EH to user.

LOCKING: None.

void **ata_eh_report**(struct ata_port * *ap*)
report error handling to user

Parameters

struct ata_port * ap ATA port to report EH about

Description

Report EH to user.

LOCKING: None.

int **ata_set_mode**(struct ata_link * *link*, struct ata_device ** *r_failed_dev*)
Program timings and issue SET FEATURES - XFER

Parameters

struct ata_link * link link on which timings will be programmed

struct ata_device ** r_failed_dev out parameter for failed device

Description

Set ATA device disk transfer mode (PIO3, UDMA6, etc.). If *ata_set_mode()* fails, pointer to the failing device is returned in **r_failed_dev**.

LOCKING: PCI/etc. bus probe sem.

Return

0 on success, negative errno otherwise

int **atapi_eh_clear_ua**(struct ata_device * *dev*)
Clear ATAPI UNIT ATTENTION after reset

Parameters

struct ata_device * dev ATAPI device to clear UA for

Description

Resets and other operations can make an ATAPI device raise UNIT ATTENTION which causes the next operation to fail. This function clears UA.

LOCKING: EH context (may sleep).

Return

0 on success, -errno on failure.

int **ata_eh_maybe_retry_flush**(struct ata_device * *dev*)
Retry FLUSH if necessary

Parameters

struct ata_device * dev ATA device which may need FLUSH retry

Description

If **dev** failed FLUSH, it needs to be reported upper layer immediately as it means that **dev** failed to remap and already lost at least a sector and further FLUSH retrials won't make any difference to the lost sector. However, if FLUSH failed for other reasons, for example transmission error, FLUSH needs to be retried.

This function determines whether FLUSH failure retry is necessary and performs it if so.

Return

0 if EH can continue, -errno if EH needs to be repeated.

int **ata_eh_set_lpm**(struct ata_link * *link*, enum ata_lpm_policy *policy*, struct ata_device ** *r_failed_dev*)
configure SATA interface power management

Parameters

struct ata_link * link link to configure power management

enum ata_lpm_policy policy the link power management policy

struct ata_device ** r_failed_dev out parameter for failed device

Description

Enable SATA Interface power management. This will enable Device Interface Power Management (DIPM) for min_power and medium_power_with_dipm policies, and then call driver specific callbacks for enabling Host Initiated Power management.

LOCKING: EH context.

Return

0 on success, -errno on failure.

int **ata_eh_recover**(struct ata_port * *ap*, ata_prereset_fn_t *prereset*, ata_reset_fn_t *softreset*, ata_reset_fn_t *hardreset*, ata_postreset_fn_t *postreset*, struct ata_link ** *r_failed_link*)
recover host port after error

Parameters

struct ata_port * ap host port to recover

ata_prereset_fn_t prereset prereset method (can be NULL)

ata_reset_fn_t softreset softreset method (can be NULL)

ata_reset_fn_t hardreset hardreset method (can be NULL)

ata_postreset_fn_t postreset postreset method (can be NULL)

struct ata_link ** r_failed_link out parameter for failed link

Description

This is the alpha and omega, eum and yang, heart and soul of libata exception handling. On entry, actions required to recover each link and hotplug requests are recorded in the link's eh_context. This function executes all the operations with appropriate retrials and fallbacks to resurrect failed devices, detach goners and greet newcomers.

LOCKING: Kernel thread context (may sleep).

Return

0 on success, -errno on failure.

void **ata_eh_finish**(struct ata_port * *ap*)
finish up EH

Parameters

struct ata_port * ap host port to finish EH for

Description

Recovery is complete. Clean up EH states and retry or finish failed qcs.

LOCKING: None.

void **ata_do_eh**(struct ata_port * *ap*, ata_prereset_fn_t *prereset*, ata_reset_fn_t *softreset*, ata_reset_fn_t *hardreset*, ata_postreset_fn_t *postreset*)
do standard error handling

Parameters

struct ata_port * ap host port to handle error for

ata_prereset_fn_t prereset prereset method (can be NULL)

ata_reset_fn_t softreset softreset method (can be NULL)

ata_reset_fn_t hardreset hardreset method (can be NULL)

ata_postreset_fn_t postreset postreset method (can be NULL)

Description

Perform standard error handling sequence.

LOCKING: Kernel thread context (may sleep).

void **ata_std_error_handler**(struct ata_port * *ap*)
standard error handler

Parameters

struct ata_port * ap host port to handle error for

Description

Standard error handler

LOCKING: Kernel thread context (may sleep).

void **ata_eh_handle_port_suspend**(struct ata_port * *ap*)
perform port suspend operation

Parameters

struct ata_port * ap port to suspend

Description

Suspend **ap**.

LOCKING: Kernel thread context (may sleep).

void **ata_eh_handle_port_resume**(struct ata_port * *ap*)
perform port resume operation

Parameters

struct ata_port * ap port to resume

Description

Resume **ap**.

LOCKING: Kernel thread context (may sleep).

libata SCSI translation/emulation

struct ata_port * **ata_sas_port_alloc**(struct ata_host * *host*, struct ata_port_info * *port_info*, struct Scsi_Host * *shost*)
Allocate port for a SAS attached SATA device

Parameters

struct ata_host * host ATA host container for all SAS ports

struct ata_port_info * port_info Information from low-level host driver

struct Scsi_Host * shost SCSI host that the scsi device is attached to

Description

LOCKING: PCI/etc. bus probe sem.

Return

ata_port pointer on success / NULL on failure.

int **ata_sas_port_start**(struct ata_port * *ap*)
Set port up for dma.

Parameters

`struct ata_port * ap` Port to initialize

Description

Called just after data structures for each port are initialized.

May be used as the `port_start()` entry in ata_port_operations.

LOCKING: Inherited from caller.

void **`ata_sas_port_stop`**(struct ata_port * *ap*)
Undo *`ata_sas_port_start()`*

Parameters

`struct ata_port * ap` Port to shut down

Description

May be used as the `port_stop()` entry in ata_port_operations.

LOCKING: Inherited from caller.

void **`ata_sas_async_probe`**(struct ata_port * *ap*)
simply schedule probing and return

Parameters

`struct ata_port * ap` Port to probe

Description

For batch scheduling of probe for sas attached ata devices, assumes the port has already been through *`ata_sas_port_init()`*

int **`ata_sas_port_init`**(struct ata_port * *ap*)
Initialize a SATA device

Parameters

`struct ata_port * ap` SATA port to initialize

Description

LOCKING: PCI/etc. bus probe sem.

Return

Zero on success, non-zero on error.

void **`ata_sas_port_destroy`**(struct ata_port * *ap*)
Destroy a SATA port allocated by ata_sas_port_alloc

Parameters

`struct ata_port * ap` SATA port to destroy

int **`ata_sas_slave_configure`**(struct scsi_device * *sdev*, struct ata_port * *ap*)
Default slave_config routine for libata devices

Parameters

`struct scsi_device * sdev` SCSI device to configure

`struct ata_port * ap` ATA port to which SCSI device is attached

Return

Zero.

int **`ata_sas_queuecmd`**(struct scsi_cmnd * *cmd*, struct ata_port * *ap*)
Issue SCSI cdb to libata-managed device

Parameters

struct scsi_cmnd * cmd SCSI command to be sent

struct ata_port * ap ATA port to which the command is being sent

Return

Return value from `__ata_scsi_queuecmd()` if **cmd** can be queued, 0 otherwise.

int **ata_std_bios_param**(struct scsi_device * *sdev*, struct block_device * *bdev*, sector_t *capacity*, int *geom*)
generic bios head/sector/cylinder calculator used by sd.

Parameters

struct scsi_device * sdev SCSI device for which BIOS geometry is to be determined

struct block_device * bdev block device associated with **sdev**

sector_t capacity capacity of SCSI device

int geom location to which geometry will be output

Description

Generic bios head/sector/cylinder calculator used by sd. Most BIOSes nowadays expect a XXX/255/16 (CHS) mapping. Some situations may arise where the disk is not bootable if this is not used.

LOCKING: Defined by the SCSI layer. We don't really care.

Return

Zero.

void **ata_scsi_unlock_native_capacity**(struct scsi_device * *sdev*)
unlock native capacity

Parameters

struct scsi_device * sdev SCSI device to adjust device capacity for

Description

This function is called if a partition on **sdev** extends beyond the end of the device. It requests EH to unlock HPA.

LOCKING: Defined by the SCSI layer. Might sleep.

int **ata_get_identity**(struct ata_port * *ap*, struct scsi_device * *sdev*, void __user * *arg*)
Handler for HDIO_GET_IDENTITY ioctl

Parameters

struct ata_port * ap target port

struct scsi_device * sdev SCSI device to get identify data for

void __user * arg User buffer area for identify data

Description

LOCKING: Defined by the SCSI layer. We don't really care.

Return

Zero on success, negative errno on error.

int **ata_cmd_ioctl**(struct scsi_device * *scsidev*, void __user * *arg*)
Handler for HDIO_DRIVE_CMD ioctl

Parameters

struct scsi_device * scsidev Device to which we are issuing command

void __user * arg User provided data for issuing command

Description

LOCKING: Defined by the SCSI layer. We don't really care.

Return

Zero on success, negative errno on error.

int **ata_task_ioctl**(struct scsi_device * *scsidev*, void __user * *arg*)
Handler for HDIO_DRIVE_TASK ioctl

Parameters

struct scsi_device * scsidev Device to which we are issuing command

void __user * arg User provided data for issuing command

Description

LOCKING: Defined by the SCSI layer. We don't really care.

Return

Zero on success, negative errno on error.

struct ata_queued_cmd * **ata_scsi_qc_new**(struct ata_device * *dev*, struct scsi_cmnd * *cmd*)
acquire new ata_queued_cmd reference

Parameters

struct ata_device * dev ATA device to which the new command is attached

struct scsi_cmnd * cmd SCSI command that originated this ATA command

Description

Obtain a reference to an unused ata_queued_cmd structure, which is the basic libata structure representing a single ATA command sent to the hardware.

If a command was available, fill in the SCSI-specific portions of the structure with information on the current command.

LOCKING: spin_lock_irqsave(host lock)

Return

Command allocated, or NULL if none available.

void **ata_dump_status**(unsigned *id*, struct ata_taskfile * *tf*)
user friendly display of error info

Parameters

unsigned id id of the port in question

struct ata_taskfile * tf ptr to filled out taskfile

Description

Decode and dump the ATA error/status registers for the user so that they have some idea what really happened at the non make-believe layer.

LOCKING: inherited from caller

void **ata_to_sense_error**(unsigned *id*, u8 *drv_stat*, u8 *drv_err*, u8 * *sk*, u8 * *asc*, u8 * *ascq*, int *verbose*)
convert ATA error to SCSI error

Parameters

unsigned id ATA device number

u8 drv_stat value contained in ATA status register

`u8 drv_err` value contained in ATA error register

`u8 * sk` the sense key we'll fill out

`u8 * asc` the additional sense code we'll fill out

`u8 * ascq` the additional sense code qualifier we'll fill out

`int verbose` be verbose

Description

Converts an ATA error into a SCSI error. Fill out pointers to SK, ASC, and ASCQ bytes for later use in fixed or descriptor format sense blocks.

LOCKING: spin_lock_irqsave(host lock)

void **`ata_gen_ata_sense`**(struct ata_queued_cmd * *qc*)
generate a SCSI fixed sense block

Parameters

`struct ata_queued_cmd * qc` Command that we are erroring out

Description

Generate sense block for a failed ATA command **qc**. Descriptor format is used to accommodate LBA48 block address.

LOCKING: None.

int **`atapi_drain_needed`**(struct request * *rq*)
Check whether data transfer may overflow

Parameters

`struct request * rq` request to be checked

Description

ATAPI commands which transfer variable length data to host might overflow due to application error or hardware bug. This function checks whether overflow should be drained and ignored for **request**.

LOCKING: None.

Return

1 if ; otherwise, 0.

int **`ata_scsi_slave_config`**(struct scsi_device * *sdev*)
Set SCSI device attributes

Parameters

`struct scsi_device * sdev` SCSI device to examine

Description

This is called before we actually start reading and writing to the device, to configure certain SCSI mid-layer behaviors.

LOCKING: Defined by SCSI layer. We don't really care.

void **`ata_scsi_slave_destroy`**(struct scsi_device * *sdev*)
SCSI device is about to be destroyed

Parameters

`struct scsi_device * sdev` SCSI device to be destroyed

Description

sdev is about to be destroyed for hot/warm unplugging. If this unplugging was initiated by libata as indicated by NULL dev->sdev, this function doesn't have to do anything. Otherwise, SCSI layer initiated warm-unplug is in progress. Clear dev->sdev, schedule the device for ATA detach and invoke EH.

LOCKING: Defined by SCSI layer. We don't really care.

int **`__ata_change_queue_depth`**(struct ata_port * *ap*, struct scsi_device * *sdev*, int *queue_depth*)
helper for ata_scsi_change_queue_depth

Parameters

`struct ata_port * ap` ATA port to which the device change the queue depth

`struct scsi_device * sdev` SCSI device to configure queue depth for

`int queue_depth` new queue depth

Description

libsas and libata have different approaches for associating a sdev to its ata_port.

int **`ata_scsi_change_queue_depth`**(struct scsi_device * *sdev*, int *queue_depth*)
SCSI callback for queue depth config

Parameters

`struct scsi_device * sdev` SCSI device to configure queue depth for

`int queue_depth` new queue depth

Description

This is libata standard hostt->change_queue_depth callback. SCSI will call into this callback when user tries to set queue depth via sysfs.

LOCKING: SCSI layer (we don't care)

Return

Newly configured queue depth.

unsigned int **`ata_scsi_start_stop_xlat`**(struct ata_queued_cmd * *qc*)
Translate SCSI START STOP UNIT command

Parameters

`struct ata_queued_cmd * qc` Storage for translated ATA taskfile

Description

Sets up an ATA taskfile to issue STANDBY (to stop) or READ VERIFY (to start). Perhaps these commands should be preceded by CHECK POWER MODE to see what power mode the device is already in. [See SAT revision 5 at www.t10.org]

LOCKING: spin_lock_irqsave(host lock)

Return

Zero on success, non-zero on error.

unsigned int **`ata_scsi_flush_xlat`**(struct ata_queued_cmd * *qc*)
Translate SCSI SYNCHRONIZE CACHE command

Parameters

`struct ata_queued_cmd * qc` Storage for translated ATA taskfile

Description

Sets up an ATA taskfile to issue FLUSH CACHE or FLUSH CACHE EXT.

LOCKING: spin_lock_irqsave(host lock)

Return

Zero on success, non-zero on error.

void **scsi_6_lba_len**(const u8 * *cdb*, u64 * *plba*, u32 * *plen*)
Get LBA and transfer length

Parameters

const u8 * cdb SCSI command to translate

u64 * plba the LBA

u32 * plen the transfer length

Description

Calculate LBA and transfer length for 6-byte commands.

void **scsi_10_lba_len**(const u8 * *cdb*, u64 * *plba*, u32 * *plen*)
Get LBA and transfer length

Parameters

const u8 * cdb SCSI command to translate

u64 * plba the LBA

u32 * plen the transfer length

Description

Calculate LBA and transfer length for 10-byte commands.

void **scsi_16_lba_len**(const u8 * *cdb*, u64 * *plba*, u32 * *plen*)
Get LBA and transfer length

Parameters

const u8 * cdb SCSI command to translate

u64 * plba the LBA

u32 * plen the transfer length

Description

Calculate LBA and transfer length for 16-byte commands.

unsigned int **ata_scsi_verify_xlat**(struct ata_queued_cmd * *qc*)
Translate SCSI VERIFY command into an ATA one

Parameters

struct ata_queued_cmd * qc Storage for translated ATA taskfile

Description

Converts SCSI VERIFY command to an ATA READ VERIFY command.

LOCKING: spin_lock_irqsave(host lock)

Return

Zero on success, non-zero on error.

unsigned int **ata_scsi_rw_xlat**(struct ata_queued_cmd * *qc*)
Translate SCSI r/w command into an ATA one

Parameters

struct ata_queued_cmd * qc Storage for translated ATA taskfile

Description

Converts any of six SCSI read/write commands into the ATA counterpart, including starting sector (LBA), sector count, and taking into account the device's LBA48 support.

Commands `READ_6`, `READ_10`, `READ_16`, `WRITE_6`, `WRITE_10`, and `WRITE_16` are currently supported.

LOCKING: spin_lock_irqsave(host lock)

Return

Zero on success, non-zero on error.

int **ata_scsi_translate**(struct ata_device * *dev*, struct scsi_cmnd * *cmd*, ata_xlat_func_t *xlat_func*)
Translate then issue SCSI command to ATA device

Parameters

struct ata_device * dev ATA device to which the command is addressed

struct scsi_cmnd * cmd SCSI command to execute

ata_xlat_func_t xlat_func Actor which translates **cmd** to an ATA taskfile

Description

Our ->:c:func:*queuecommand()* function has decided that the SCSI command issued can be directly translated into an ATA command, rather than handled internally.

This function sets up an ata_queued_cmd structure for the SCSI command, and sends that ata_queued_cmd to the hardware.

The xlat_func argument (actor) returns 0 if ready to execute ATA command, else 1 to finish translation. If 1 is returned then cmd->result (and possibly cmd->sense_buffer) are assumed to be set reflecting an error condition or clean (early) termination.

LOCKING: spin_lock_irqsave(host lock)

Return

0 on success, SCSI_ML_QUEUE_DEVICE_BUSY if the command needs to be deferred.

void * **ata_scsi_rbuf_get**(struct scsi_cmnd * *cmd*, bool *copy_in*, unsigned long * *flags*)
Map response buffer.

Parameters

struct scsi_cmnd * cmd SCSI command containing buffer to be mapped.

bool copy_in copy in from user buffer

unsigned long * flags unsigned long variable to store irq enable status

Description

Prepare buffer for simulated SCSI commands.

LOCKING: spin_lock_irqsave(ata_scsi_rbuf_lock) on success

Return

Pointer to response buffer.

void **ata_scsi_rbuf_put**(struct scsi_cmnd * *cmd*, bool *copy_out*, unsigned long * *flags*)
Unmap response buffer.

Parameters

struct scsi_cmnd * cmd SCSI command containing buffer to be unmapped.

bool copy_out copy out result

unsigned long * flags flags passed to *ata_scsi_rbuf_get()*

Description

Returns rbuf buffer. The result is copied to **cmd**'s buffer if **copy_back** is true.

LOCKING: Unlocks ata_scsi_rbuf_lock.

void **ata_scsi_rbuf_fill**(struct ata_scsi_args * *args*, unsigned int (*actor) (struct ata_scsi_args **args*, u8 **rbuf*)
wrapper for SCSI command simulators

Parameters

struct ata_scsi_args * args device IDENTIFY data / SCSI command of interest.

unsigned int (*)(struct ata_scsi_args *args, u8 *rbuf) actor Callback hook for desired SCSI command simulator

Description

Takes care of the hard work of simulating a SCSI command... Mapping the response buffer, calling the command's handler, and handling the handler's return value. This return value indicates whether the handler wishes the SCSI command to be completed successfully (0), or not (in which case cmd->result and sense buffer are assumed to be set).

LOCKING: spin_lock_irqsave(host lock)

unsigned int **ata_scsiop_inq_std**(struct ata_scsi_args * *args*, u8 * *rbuf*)
Simulate INQUIRY command

Parameters

struct ata_scsi_args * args device IDENTIFY data / SCSI command of interest.

u8 * rbuf Response buffer, to which simulated SCSI cmd output is sent.

Description

Returns standard device identification data associated with non-VPD INQUIRY command output.

LOCKING: spin_lock_irqsave(host lock)

unsigned int **ata_scsiop_inq_00**(struct ata_scsi_args * *args*, u8 * *rbuf*)
Simulate INQUIRY VPD page 0, list of pages

Parameters

struct ata_scsi_args * args device IDENTIFY data / SCSI command of interest.

u8 * rbuf Response buffer, to which simulated SCSI cmd output is sent.

Description

Returns list of inquiry VPD pages available.

LOCKING: spin_lock_irqsave(host lock)

unsigned int **ata_scsiop_inq_80**(struct ata_scsi_args * *args*, u8 * *rbuf*)
Simulate INQUIRY VPD page 80, device serial number

Parameters

struct ata_scsi_args * args device IDENTIFY data / SCSI command of interest.

u8 * rbuf Response buffer, to which simulated SCSI cmd output is sent.

Description

Returns ATA device serial number.

LOCKING: spin_lock_irqsave(host lock)

unsigned int **ata_scsiop_inq_83**(struct ata_scsi_args * *args*, u8 * *rbuf*)
Simulate INQUIRY VPD page 83, device identity

Parameters

`struct ata_scsi_args * args` device IDENTIFY data / SCSI command of interest.

`u8 * rbuf` Response buffer, to which simulated SCSI cmd output is sent.

Description

Yields two logical unit device identification designators:

- vendor specific ASCII containing the ATA serial number
- SAT defined "t10 vendor id based" containing ASCII vendor name ("ATA "), model and serial numbers.

LOCKING: spin_lock_irqsave(host lock)

unsigned int **`ata_scsiop_inq_89`**(struct ata_scsi_args * *args*, u8 * *rbuf*)
Simulate INQUIRY VPD page 89, ATA info

Parameters

`struct ata_scsi_args * args` device IDENTIFY data / SCSI command of interest.

`u8 * rbuf` Response buffer, to which simulated SCSI cmd output is sent.

Description

Yields SAT-specified ATA VPD page.

LOCKING: spin_lock_irqsave(host lock)

void **`modecpy`**(u8 * *dest*, const u8 * *src*, int *n*, bool *changeable*)
Prepare response for MODE SENSE

Parameters

`u8 * dest` output buffer

`const u8 * src` data being copied

`int n` length of mode page

`bool changeable` whether changeable parameters are requested

Description

Generate a generic MODE SENSE page for either current or changeable parameters.

LOCKING: None.

unsigned int **`ata_msense_caching`**(u16 * *id*, u8 * *buf*, bool *changeable*)
Simulate MODE SENSE caching info page

Parameters

`u16 * id` device IDENTIFY data

`u8 * buf` output buffer

`bool changeable` whether changeable parameters are requested

Description

Generate a caching info page, which conditionally indicates write caching to the SCSI layer, depending on device capabilities.

LOCKING: None.

unsigned int **`ata_msense_control`**(struct ata_device * *dev*, u8 * *buf*, bool *changeable*)
Simulate MODE SENSE control mode page

Parameters

`struct ata_device * dev` ATA device of interest

u8 * buf output buffer

bool changeable whether changeable parameters are requested

Description

Generate a generic MODE SENSE control mode page.

LOCKING: None.

unsigned int **ata_msense_rw_recovery**(u8 * *buf*, bool *changeable*)
Simulate MODE SENSE r/w error recovery page

Parameters

u8 * buf output buffer

bool changeable whether changeable parameters are requested

Description

Generate a generic MODE SENSE r/w error recovery page.

LOCKING: None.

unsigned int **ata_scsiop_mode_sense**(struct ata_scsi_args * *args*, u8 * *rbuf*)
Simulate MODE SENSE 6, 10 commands

Parameters

struct ata_scsi_args * args device IDENTIFY data / SCSI command of interest.

u8 * rbuf Response buffer, to which simulated SCSI cmd output is sent.

Description

Simulate MODE SENSE commands. Assume this is invoked for direct access devices (e.g. disks) only. There should be no block descriptor for other device types.

LOCKING: spin_lock_irqsave(host lock)

unsigned int **ata_scsiop_read_cap**(struct ata_scsi_args * *args*, u8 * *rbuf*)
Simulate READ CAPACITY[16] commands

Parameters

struct ata_scsi_args * args device IDENTIFY data / SCSI command of interest.

u8 * rbuf Response buffer, to which simulated SCSI cmd output is sent.

Description

Simulate READ CAPACITY commands.

LOCKING: None.

unsigned int **ata_scsiop_report_luns**(struct ata_scsi_args * *args*, u8 * *rbuf*)
Simulate REPORT LUNS command

Parameters

struct ata_scsi_args * args device IDENTIFY data / SCSI command of interest.

u8 * rbuf Response buffer, to which simulated SCSI cmd output is sent.

Description

Simulate REPORT LUNS command.

LOCKING: spin_lock_irqsave(host lock)

unsigned int **atapi_xlat**(struct ata_queued_cmd * *qc*)
Initialize PACKET taskfile

Parameters

struct ata_queued_cmd * qc command structure to be initialized

Description

LOCKING: spin_lock_irqsave(host lock)

Return

Zero on success, non-zero on failure.

struct ata_device * **ata_scsi_find_dev**(struct ata_port * *ap*, const struct scsi_device * *scsidev*)
lookup ata_device from scsi_cmnd

Parameters

struct ata_port * ap ATA port to which the device is attached

const struct scsi_device * scsidev SCSI device from which we derive the ATA device

Description

Given various information provided in struct scsi_cmnd, map that onto an ATA bus, and using that mapping determine which ata_device is associated with the SCSI command to be sent.

LOCKING: spin_lock_irqsave(host lock)

Return

Associated ATA device, or `NULL` if not found.

unsigned int **ata_scsi_pass_thru**(struct ata_queued_cmd * *qc*)
convert ATA pass-thru CDB to taskfile

Parameters

struct ata_queued_cmd * qc command structure to be initialized

Description

Handles either 12, 16, or 32-byte versions of the CDB.

Return

Zero on success, non-zero on failure.

size_t **ata_format_dsm_trim_descr**(struct scsi_cmnd * *cmd*, u32 *trmax*, u64 *sector*, u32 *count*)
SATL Write Same to DSM Trim

Parameters

struct scsi_cmnd * cmd SCSI command being translated

u32 trmax Maximum number of entries that will fit in sector_size bytes.

u64 sector Starting sector

u32 count Total Range of request in logical sectors

Description

Rewrite the WRITE SAME descriptor to be a DSM TRIM little-endian formatted descriptor.

Upto 64 entries of the format:

63:48 Range Length 47:0 LBA

Range Length of 0 is ignored. LBA's should be sorted order and not overlap.

NOTE

this is the same format as ADD LBA(S) TO NV CACHE PINNED SET

Return

Number of bytes copied into sglist.

unsigned int **ata_scsi_write_same_xlat**(struct ata_queued_cmd * *qc*)
SATL Write Same to ATA SCT Write Same

Parameters

struct ata_queued_cmd * qc Command to be translated

Description

Translate a SCSI WRITE SAME command to be either a DSM TRIM command or an SCT Write Same command. Based on WRITE SAME has the UNMAP flag:

- When set translate to DSM TRIM
- When clear translate to SCT Write Same

unsigned int **ata_scsiop_maint_in**(struct ata_scsi_args * *args*, u8 * *rbuf*)
Simulate a subset of MAINTENANCE_IN

Parameters

struct ata_scsi_args * args device MAINTENANCE_IN data / SCSI command of interest.

u8 * rbuf Response buffer, to which simulated SCSI cmd output is sent.

Description

Yields a subset to satisfy *scsi_report_opcode()*

LOCKING: spin_lock_irqsave(host lock)

void **ata_scsi_report_zones_complete**(struct ata_queued_cmd * *qc*)
convert ATA output

Parameters

struct ata_queued_cmd * qc command structure returning the data

Description

Convert T-13 little-endian field representation into T-10 big-endian field representation. What a mess.

int **ata_mselect_caching**(struct ata_queued_cmd * *qc*, const u8 * *buf*, int *len*, u16 * *fp*)
Simulate MODE SELECT for caching info page

Parameters

struct ata_queued_cmd * qc Storage for translated ATA taskfile

const u8 * buf input buffer

int len number of valid bytes in the input buffer

u16 * fp out parameter for the failed field on error

Description

Prepare a taskfile to modify caching information for the device.

LOCKING: None.

int **ata_mselect_control**(struct ata_queued_cmd * *qc*, const u8 * *buf*, int *len*, u16 * *fp*)
Simulate MODE SELECT for control page

Parameters

struct ata_queued_cmd * qc Storage for translated ATA taskfile

const u8 * buf input buffer

int len number of valid bytes in the input buffer

u16 * fp out parameter for the failed field on error

Description

Prepare a taskfile to modify caching information for the device.

LOCKING: None.

unsigned int **ata_scsi_mode_select_xlat**(struct ata_queued_cmd * *qc*)
Simulate MODE SELECT 6, 10 commands

Parameters

struct ata_queued_cmd * qc Storage for translated ATA taskfile

Description

Converts a MODE SELECT command to an ATA SET FEATURES taskfile. Assume this is invoked for direct access devices (e.g. disks) only. There should be no block descriptor for other device types.

LOCKING: spin_lock_irqsave(host lock)

unsigned int **ata_scsi_var_len_cdb_xlat**(struct ata_queued_cmd * *qc*)
SATL variable length CDB to Handler

Parameters

struct ata_queued_cmd * qc Command to be translated

Description

Translate a SCSI variable length CDB to specified commands. It checks a service action value in CDB to call corresponding handler.

Return

Zero on success, non-zero on failure

ata_xlat_func_t **ata_get_xlat_func**(struct ata_device * *dev*, u8 *cmd*)
check if SCSI to ATA translation is possible

Parameters

struct ata_device * dev ATA device

u8 cmd SCSI command opcode to consider

Description

Look up the SCSI command given, and determine whether the SCSI command is to be translated or simulated.

Return

Pointer to translation function if possible, `NULL` if not.

void **ata_scsi_dump_cdb**(struct ata_port * *ap*, struct scsi_cmnd * *cmd*)
dump SCSI command contents to dmesg

Parameters

struct ata_port * ap ATA port to which the command was being sent

struct scsi_cmnd * cmd SCSI command to dump

Description

Prints the contents of a SCSI command via *printk()*.

int **ata_scsi_queuecmd**(struct Scsi_Host * *shost*, struct scsi_cmnd * *cmd*)
Issue SCSI cdb to libata-managed device

Parameters

struct Scsi_Host * shost SCSI host of command to be sent

`struct scsi_cmnd * cmd` SCSI command to be sent

Description

In some cases, this function translates SCSI commands into ATA taskfiles, and queues the taskfiles to be sent to hardware. In other cases, this function simulates a SCSI device by evaluating and responding to certain SCSI commands. This creates the overall effect of ATA and ATAPI devices appearing as SCSI devices.

LOCKING: ATA host lock

Return

Return value from `__ata_scsi_queuecmd()` if **cmd** can be queued, 0 otherwise.

void **`ata_scsi_simulate`**(struct ata_device * *dev*, struct scsi_cmnd * *cmd*)
simulate SCSI command on ATA device

Parameters

`struct ata_device * dev` the target device

`struct scsi_cmnd * cmd` SCSI command being sent to device.

Description

Interprets and directly executes a select list of SCSI commands that can be handled internally.

LOCKING: spin_lock_irqsave(host lock)

int **`ata_scsi_offline_dev`**(struct ata_device * *dev*)
offline attached SCSI device

Parameters

`struct ata_device * dev` ATA device to offline attached SCSI device for

Description

This function is called from `ata_eh_hotplug()` and responsible for taking the SCSI device attached to **dev** offline. This function is called with host lock which protects dev->sdev against clearing.

LOCKING: spin_lock_irqsave(host lock)

Return

1 if attached SCSI device exists, 0 otherwise.

void **`ata_scsi_remove_dev`**(struct ata_device * *dev*)
remove attached SCSI device

Parameters

`struct ata_device * dev` ATA device to remove attached SCSI device for

Description

This function is called from `ata_eh_scsi_hotplug()` and responsible for removing the SCSI device attached to **dev**.

LOCKING: Kernel thread context (may sleep).

void **`ata_scsi_media_change_notify`**(struct ata_device * *dev*)
send media change event

Parameters

`struct ata_device * dev` Pointer to the disk device with media change event

Description

Tell the block layer to send a media change notification event.

LOCKING: spin_lock_irqsave(host lock)

void **ata_scsi_hotplug**(struct work_struct * *work*)
SCSI part of hotplug

Parameters

struct work_struct * work Pointer to ATA port to perform SCSI hotplug on

Description

Perform SCSI part of hotplug. It's executed from a separate workqueue after EH completes. This is necessary because SCSI hot plugging requires working EH and hot unplugging is synchronized with hot plugging with a mutex.

LOCKING: Kernel thread context (may sleep).

int **ata_scsi_user_scan**(struct Scsi_Host * *shost*, unsigned int *channel*, unsigned int *id*, u64 *lun*)
indication for user-initiated bus scan

Parameters

struct Scsi_Host * shost SCSI host to scan

unsigned int channel Channel to scan

unsigned int id ID to scan

u64 lun LUN to scan

Description

This function is called when user explicitly requests bus scan. Set probe pending flag and invoke EH.

LOCKING: SCSI layer (we don't care)

Return

Zero.

void **ata_scsi_dev_rescan**(struct work_struct * *work*)
initiate `scsi_rescan_device()`

Parameters

struct work_struct * work Pointer to ATA port to perform `scsi_rescan_device()`

Description

After ATA pass thru (SAT) commands are executed successfully, libata need to propagate the changes to SCSI layer.

LOCKING: Kernel thread context (may sleep).

ATA errors and exceptions

This chapter tries to identify what error/exception conditions exist for ATA/ATAPI devices and describe how they should be handled in implementation-neutral way.

The term 'error' is used to describe conditions where either an explicit error condition is reported from device or a command has timed out.

The term 'exception' is either used to describe exceptional conditions which are not errors (say, power or hotplug events), or to describe both errors and non-error exceptional conditions. Where explicit distinction between error and exception is necessary, the term 'non-error exception' is used.

Exception categories

Exceptions are described primarily with respect to legacy taskfile + bus master IDE interface. If a controller provides other better mechanism for error reporting, mapping those into categories described below shouldn't be difficult.

In the following sections, two recovery actions - reset and reconfiguring transport - are mentioned. These are described further in *EH recovery actions*.

HSM violation

This error is indicated when STATUS value doesn't match HSM requirement during issuing or execution any ATA/ATAPI command.

- ATA_STATUS doesn't contain !BSY && DRDY && !DRQ while trying to issue a command.
- !BSY && !DRQ during PIO data transfer.
- DRQ on command completion.
- !BSY && ERR after CDB transfer starts but before the last byte of CDB is transferred. ATA/ATAPI standard states that "The device shall not terminate the PACKET command with an error before the last byte of the command packet has been written" in the error outputs description of PACKET command and the state diagram doesn't include such transitions.

In these cases, HSM is violated and not much information regarding the error can be acquired from STATUS or ERROR register. IOW, this error can be anything - driver bug, faulty device, controller and/or cable.

As HSM is violated, reset is necessary to restore known state. Reconfiguring transport for lower speed might be helpful too as transmission errors sometimes cause this kind of errors.

ATA/ATAPI device error (non-NCQ / non-CHECK CONDITION)

These are errors detected and reported by ATA/ATAPI devices indicating device problems. For this type of errors, STATUS and ERROR register values are valid and describe error condition. Note that some of ATA bus errors are detected by ATA/ATAPI devices and reported using the same mechanism as device errors. Those cases are described later in this section.

For ATA commands, this type of errors are indicated by !BSY && ERR during command execution and on completion.

For ATAPI commands,

- !BSY && ERR && ABRT right after issuing PACKET indicates that PACKET command is not supported and falls in this category.
- !BSY && ERR(==CHK) && !ABRT after the last byte of CDB is transferred indicates CHECK CONDITION and doesn't fall in this category.
- !BSY && ERR(==CHK) && ABRT after the last byte of CDB is transferred *probably* indicates CHECK CONDITION and doesn't fall in this category.

Of errors detected as above, the following are not ATA/ATAPI device errors but ATA bus errors and should be handled according to *ATA bus error*.

CRC error during data transfer This is indicated by ICRC bit in the ERROR register and means that corruption occurred during data transfer. Up to ATA/ATAPI-7, the standard specifies that this bit is only applicable to UDMA transfers but ATA/ATAPI-8 draft revision 1f says that the bit may be applicable to multiword DMA and PIO.

ABRT error during data transfer or on completion Up to ATA/ATAPI-7, the standard specifies that ABRT could be set on ICRC errors and on cases where a device is not able to complete a command. Combined with the fact that MWDMA and PIO transfer errors aren't allowed to use ICRC bit up to ATA/ATAPI-7, it seems to imply that ABRT bit alone could indicate transfer errors.

However, ATA/ATAPI-8 draft revision 1f removes the part that ICRC errors can turn on ABRT. So, this is kind of gray area. Some heuristics are needed here.

ATA/ATAPI device errors can be further categorized as follows.

Media errors This is indicated by UNC bit in the ERROR register. ATA devices reports UNC error only after certain number of retries cannot recover the data, so there's nothing much else to do other than notifying upper layer.

READ and WRITE commands report CHS or LBA of the first failed sector but ATA/ATAPI standard specifies that the amount of transferred data on error completion is indeterminate, so we cannot assume that sectors preceding the failed sector have been transferred and thus cannot complete those sectors successfully as SCSI does.

Media changed / media change requested error <<TODO: fill here>>

Address error This is indicated by IDNF bit in the ERROR register. Report to upper layer.

Other errors This can be invalid command or parameter indicated by ABRT ERROR bit or some other error condition. Note that ABRT bit can indicate a lot of things including ICRC and Address errors. Heuristics needed.

Depending on commands, not all STATUS/ERROR bits are applicable. These non-applicable bits are marked with "na" in the output descriptions but up to ATA/ATAPI-7 no definition of "na" can be found. However, ATA/ATAPI-8 draft revision 1f describes "N/A" as follows.

> **3.2.3.3a N/A** A keyword the indicates a field has no defined value in this standard and should not be checked by the host or device. N/A fields should be cleared to zero.

So, it seems reasonable to assume that "na" bits are cleared to zero by devices and thus need no explicit masking.

ATAPI device CHECK CONDITION

ATAPI device CHECK CONDITION error is indicated by set CHK bit (ERR bit) in the STATUS register after the last byte of CDB is transferred for a PACKET command. For this kind of errors, sense data should be acquired to gather information regarding the errors. REQUEST SENSE packet command should be used to acquire sense data.

Once sense data is acquired, this type of errors can be handled similarly to other SCSI errors. Note that sense data may indicate ATA bus error (e.g. Sense Key 04h HARDWARE ERROR && ASC/ASCQ 47h/00h SCSI PARITY ERROR). In such cases, the error should be considered as an ATA bus error and handled according to *ATA bus error*.

ATA device error (NCQ)

NCQ command error is indicated by cleared BSY and set ERR bit during NCQ command phase (one or more NCQ commands outstanding). Although STATUS and ERROR registers will contain valid values describing the error, READ LOG EXT is required to clear the error condition, determine which command has failed and acquire more information.

READ LOG EXT Log Page 10h reports which tag has failed and taskfile register values describing the error. With this information the failed command can be handled as a normal ATA command error as in *ATA/ATAPI device error (non-NCQ / non-CHECK CONDITION)* and all other in-flight commands must be retried. Note that this retry should not be counted - it's likely that commands retried this way would have completed normally if it were not for the failed command.

Note that ATA bus errors can be reported as ATA device NCQ errors. This should be handled as described in *ATA bus error*.

If READ LOG EXT Log Page 10h fails or reports NQ, we're thoroughly screwed. This condition should be treated according to *HSM violation*.

ATA bus error

ATA bus error means that data corruption occurred during transmission over ATA bus (SATA or PATA). This type of errors can be indicated by

- ICRC or ABRT error as described in *ATA/ATAPI device error (non-NCQ / non-CHECK CONDITION)*.
- Controller-specific error completion with error information indicating transmission error.
- On some controllers, command timeout. In this case, there may be a mechanism to determine that the timeout is due to transmission error.
- Unknown/random errors, timeouts and all sorts of weirdities.

As described above, transmission errors can cause wide variety of symptoms ranging from device ICRC error to random device lockup, and, for many cases, there is no way to tell if an error condition is due to transmission error or not; therefore, it's necessary to employ some kind of heuristic when dealing with errors and timeouts. For example, encountering repetitive ABRT errors for known supported command is likely to indicate ATA bus error.

Once it's determined that ATA bus errors have possibly occurred, lowering ATA bus transmission speed is one of actions which may alleviate the problem. See *Reconfigure transport* for more information.

PCI bus error

Data corruption or other failures during transmission over PCI (or other system bus). For standard BMDMA, this is indicated by Error bit in the BMDMA Status register. This type of errors must be logged as it indicates something is very wrong with the system. Resetting host controller is recommended.

Late completion

This occurs when timeout occurs and the timeout handler finds out that the timed out command has completed successfully or with error. This is usually caused by lost interrupts. This type of errors must be logged. Resetting host controller is recommended.

Unknown error (timeout)

This is when timeout occurs and the command is still processing or the host and device are in unknown state. When this occurs, HSM could be in any valid or invalid state. To bring the device to known state and make it forget about the timed out command, resetting is necessary. The timed out command may be retried.

Timeouts can also be caused by transmission errors. Refer to *ATA bus error* for more details.

Hotplug and power management exceptions

<<TODO: fill here>>

EH recovery actions

This section discusses several important recovery actions.

Clearing error condition

Many controllers require its error registers to be cleared by error handler. Different controllers may have different requirements.

For SATA, it's strongly recommended to clear at least SError register during error handling.

Reset

During EH, resetting is necessary in the following cases.

- HSM is in unknown or invalid state
- HBA is in unknown or invalid state
- EH needs to make HBA/device forget about in-flight commands
- HBA/device behaves weirdly

Resetting during EH might be a good idea regardless of error condition to improve EH robustness. Whether to reset both or either one of HBA and device depends on situation but the following scheme is recommended.

- When it's known that HBA is in ready state but ATA/ATAPI device is in unknown state, reset only device.
- If HBA is in unknown state, reset both HBA and device.

HBA resetting is implementation specific. For a controller complying to taskfile/BMDMA PCI IDE, stopping active DMA transaction may be sufficient iff BMDMA state is the only HBA context. But even mostly taskfile/BMDMA PCI IDE complying controllers may have implementation specific requirements and mechanism to reset themselves. This must be addressed by specific drivers.

OTOH, ATA/ATAPI standard describes in detail ways to reset ATA/ATAPI devices.

PATA hardware reset This is hardware initiated device reset signalled with asserted PATA RESET- signal. There is no standard way to initiate hardware reset from software although some hardware provides registers that allow driver to directly tweak the RESET- signal.

Software reset This is achieved by turning CONTROL SRST bit on for at least 5us. Both PATA and SATA support it but, in case of SATA, this may require controller-specific support as the second Register FIS to clear SRST should be transmitted while BSY bit is still set. Note that on PATA, this resets both master and slave devices on a channel.

EXECUTE DEVICE DIAGNOSTIC command Although ATA/ATAPI standard doesn't describe exactly, EDD implies some level of resetting, possibly similar level with software reset. Host-side EDD protocol can be handled with normal command processing and most SATA controllers should be able to handle EDD's just like other commands. As in software reset, EDD affects both devices on a PATA bus.

Although EDD does reset devices, this doesn't suit error handling as EDD cannot be issued while BSY is set and it's unclear how it will act when device is in unknown/weird state.

ATAPI DEVICE RESET command This is very similar to software reset except that reset can be restricted to the selected device without affecting the other device sharing the cable.

SATA phy reset This is the preferred way of resetting a SATA device. In effect, it's identical to PATA hardware reset. Note that this can be done with the standard SCR Control register. As such, it's usually easier to implement than software reset.

One more thing to consider when resetting devices is that resetting clears certain configuration parameters and they need to be set to their previous or newly adjusted values after reset.

Parameters affected are.

- CHS set up with INITIALIZE DEVICE PARAMETERS (seldom used)
- Parameters set with SET FEATURES including transfer mode setting
- Block count set with SET MULTIPLE MODE
- Other parameters (SET MAX, MEDIA LOCK...)

ATA/ATAPI standard specifies that some parameters must be maintained across hardware or software reset, but doesn't strictly specify all of them. Always reconfiguring needed parameters after reset is required for robustness. Note that this also applies when resuming from deep sleep (power-off).

Also, ATA/ATAPI standard requires that IDENTIFY DEVICE / IDENTIFY PACKET DEVICE is issued after any configuration parameter is updated or a hardware reset and the result used for further operation. OS driver is required to implement revalidation mechanism to support this.

Reconfigure transport

For both PATA and SATA, a lot of corners are cut for cheap connectors, cables or controllers and it's quite common to see high transmission error rate. This can be mitigated by lowering transmission speed.

The following is a possible scheme Jeff Garzik suggested.

> If more than $N (3?) transmission errors happen in 15 minutes,
>
> - if SATA, decrease SATA PHY speed. if speed cannot be decreased,
> - decrease UDMA xfer speed. if at UDMA0, switch to PIO4,
> - decrease PIO xfer speed. if at PIO3, complain, but continue

ata_piix Internals

int **`ich_pata_cable_detect`**(struct ata_port * *ap*)
Probe host controller cable detect info

Parameters

`struct ata_port * ap` Port for which cable detect info is desired

Description

Read 80c cable indicator from ATA PCI device's PCI config register. This register is normally set by firmware (BIOS).

LOCKING: None (inherited from caller).

int **`piix_pata_prereset`**(struct ata_link * *link*, unsigned long *deadline*)
prereset for PATA host controller

Parameters

`struct ata_link * link` Target link

`unsigned long deadline` deadline jiffies for the operation

Description

LOCKING: None (inherited from caller).

void **`piix_set_piomode`**(struct ata_port * *ap*, struct ata_device * *adev*)
Initialize host controller PATA PIO timings

Parameters

`struct ata_port * ap` Port whose timings we are configuring

`struct ata_device * adev` Drive in question

Description

Set PIO mode for device, in host controller PCI config space.

LOCKING: None (inherited from caller).

void **`do_pata_set_dmamode`**(struct ata_port * *ap*, struct ata_device * *adev*, int *isich*)
Initialize host controller PATA PIO timings

Parameters

`struct ata_port * ap` Port whose timings we are configuring

`struct ata_device * adev` Drive in question

`int isich` set if the chip is an ICH device

Description

Set UDMA mode for device, in host controller PCI config space.

LOCKING: None (inherited from caller).

void **`piix_set_dmamode`**(struct ata_port * *ap*, struct ata_device * *adev*)
Initialize host controller PATA DMA timings

Parameters

`struct ata_port * ap` Port whose timings we are configuring

`struct ata_device * adev` um

Description

Set MW/UDMA mode for device, in host controller PCI config space.

LOCKING: None (inherited from caller).

void **`ich_set_dmamode`**(struct ata_port * *ap*, struct ata_device * *adev*)
Initialize host controller PATA DMA timings

Parameters

`struct ata_port * ap` Port whose timings we are configuring

`struct ata_device * adev` um

Description

Set MW/UDMA mode for device, in host controller PCI config space.

LOCKING: None (inherited from caller).

int **`piix_check_450nx_errata`**(struct pci_dev * *ata_dev*)
Check for problem 450NX setup

Parameters

`struct pci_dev * ata_dev` the PCI device to check

Description

Check for the present of 450NX errata #19 and errata #25. If they are found return an error code so we can turn off DMA

int **`piix_init_one`**(struct pci_dev * *pdev*, const struct pci_device_id * *ent*)
Register PIIX ATA PCI device with kernel services

Parameters

`struct pci_dev * pdev` PCI device to register

`const struct pci_device_id * ent` Entry in piix_pci_tbl matching with **pdev**

Description

Called from kernel PCI layer. We probe for combined mode (sigh), and then hand over control to libata, for it to do the rest.

LOCKING: Inherited from PCI layer (may sleep).

Return

Zero on success, or -ERRNO value.

sata_sil Internals

int **sil_set_mode**(struct ata_link * *link*, struct ata_device ** *r_failed*)
wrap set_mode functions

Parameters

struct ata_link * link link to set up

struct ata_device ** r_failed returned device when we fail

Description

Wrap the libata method for device setup as after the setup we need to inspect the results and do some configuration work

void **sil_dev_config**(struct ata_device * *dev*)
Apply device/host-specific errata fixups

Parameters

struct ata_device * dev Device to be examined

Description

After the IDENTIFY [PACKET] DEVICE step is complete, and a device is known to be present, this function is called. We apply two errata fixups which are specific to Silicon Image, a Seagate and a Maxtor fixup.

For certain Seagate devices, we must limit the maximum sectors to under 8K.

For certain Maxtor devices, we must not program the drive beyond udma5.

Both fixups are unfairly pessimistic. As soon as I get more information on these errata, I will create a more exhaustive list, and apply the fixups to only the specific devices/hosts/firmwares that need it.

20040111 - Seagate drives affected by the Mod15Write bug are blacklisted The Maxtor quirk is in the blacklist, but I'm keeping the original pessimistic fix for the following reasons… - There seems to be less info on it, only one device gleaned off the Windows driver, maybe only one is affected. More info would be greatly appreciated. - But then again UDMA5 is hardly anything to complain about

Thanks

The bulk of the ATA knowledge comes thanks to long conversations with Andre Hedrick (www.linux-ide.org), and long hours pondering the ATA and SCSI specifications.

Thanks to Alan Cox for pointing out similarities between SATA and SCSI, and in general for motivation to hack on libata.

libata's device detection method, ata_pio_devchk, and in general all the early probing was based on extensive study of Hale Landis's probe/reset code in his ATADRVR driver (www.ata-atapi.com).

CHAPTER

TWENTYTWO

MTD NAND DRIVER PROGRAMMING INTERFACE

Author Thomas Gleixner

Introduction

The generic NAND driver supports almost all NAND and AG-AND based chips and connects them to the Memory Technology Devices (MTD) subsystem of the Linux Kernel.

This documentation is provided for developers who want to implement board drivers or filesystem drivers suitable for NAND devices.

Known Bugs And Assumptions

None.

Documentation hints

The function and structure docs are autogenerated. Each function and struct member has a short description which is marked with an [XXX] identifier. The following chapters explain the meaning of those identifiers.

Function identifiers [XXX]

The functions are marked with [XXX] identifiers in the short comment. The identifiers explain the usage and scope of the functions. Following identifiers are used:

- [MTD Interface]

  These functions provide the interface to the MTD kernel API. They are not replaceable and provide functionality which is complete hardware independent.

- [NAND Interface]

  These functions are exported and provide the interface to the NAND kernel API.

- [GENERIC]

  Generic functions are not replaceable and provide functionality which is complete hardware independent.

- [DEFAULT]

  Default functions provide hardware related functionality which is suitable for most of the implementations. These functions can be replaced by the board driver if necessary. Those functions are called via pointers in the NAND chip description structure. The board driver can set the functions which should be replaced by board dependent functions before calling nand_scan(). If the function pointer is NULL on entry to nand_scan() then the pointer is set to the default function which is suitable for the detected chip type.

Struct member identifiers [XXX]

The struct members are marked with [XXX] identifiers in the comment. The identifiers explain the usage and scope of the members. Following identifiers are used:

- [INTERN]

  These members are for NAND driver internal use only and must not be modified. Most of these values are calculated from the chip geometry information which is evaluated during nand_scan().

- [REPLACEABLE]

  Replaceable members hold hardware related functions which can be provided by the board driver. The board driver can set the functions which should be replaced by board dependent functions before calling nand_scan(). If the function pointer is NULL on entry to nand_scan() then the pointer is set to the default function which is suitable for the detected chip type.

- [BOARDSPECIFIC]

  Board specific members hold hardware related information which must be provided by the board driver. The board driver must set the function pointers and datafields before calling nand_scan().

- [OPTIONAL]

  Optional members can hold information relevant for the board driver. The generic NAND driver code does not use this information.

Basic board driver

For most boards it will be sufficient to provide just the basic functions and fill out some really board dependent members in the nand chip description structure.

Basic defines

At least you have to provide a nand_chip structure and a storage for the ioremap'ed chip address. You can allocate the nand_chip structure using kmalloc or you can allocate it statically. The NAND chip structure embeds an mtd structure which will be registered to the MTD subsystem. You can extract a pointer to the mtd structure from a nand_chip pointer using the nand_to_mtd() helper.

Kmalloc based example

```
static struct mtd_info *board_mtd;
static void __iomem *baseaddr;
```

Static example

```
static struct nand_chip board_chip;
static void __iomem *baseaddr;
```

Partition defines

If you want to divide your device into partitions, then define a partitioning scheme suitable to your board.

```
#define NUM_PARTITIONS 2
static struct mtd_partition partition_info[] = {
    { .name = "Flash partition 1",
      .offset =  0,
      .size =    8 * 1024 * 1024 },
    { .name = "Flash partition 2",
      .offset =  MTDPART_OFS_NEXT,
      .size =    MTDPART_SIZ_FULL },
};
```

Hardware control function

The hardware control function provides access to the control pins of the NAND chip(s). The access can be done by GPIO pins or by address lines. If you use address lines, make sure that the timing requirements are met.

GPIO based example

```
static void board_hwcontrol(struct mtd_info *mtd, int cmd)
{
    switch(cmd){
        case NAND_CTL_SETCLE: /* Set CLE pin high */ break;
        case NAND_CTL_CLRCLE: /* Set CLE pin low */ break;
        case NAND_CTL_SETALE: /* Set ALE pin high */ break;
        case NAND_CTL_CLRALE: /* Set ALE pin low */ break;
        case NAND_CTL_SETNCE: /* Set nCE pin low */ break;
        case NAND_CTL_CLRNCE: /* Set nCE pin high */ break;
    }
}
```

Address lines based example. It's assumed that the nCE pin is driven by a chip select decoder.

```
static void board_hwcontrol(struct mtd_info *mtd, int cmd)
{
    struct nand_chip *this = mtd_to_nand(mtd);
    switch(cmd){
        case NAND_CTL_SETCLE: this->IO_ADDR_W |= CLE_ADRR_BIT;  break;
        case NAND_CTL_CLRCLE: this->IO_ADDR_W &= ~CLE_ADRR_BIT; break;
        case NAND_CTL_SETALE: this->IO_ADDR_W |= ALE_ADRR_BIT;  break;
        case NAND_CTL_CLRALE: this->IO_ADDR_W &= ~ALE_ADRR_BIT; break;
    }
}
```

Device ready function

If the hardware interface has the ready busy pin of the NAND chip connected to a GPIO or other accessible I/O pin, this function is used to read back the state of the pin. The function has no arguments and should return 0, if the device is busy (R/B pin is low) and 1, if the device is ready (R/B pin is high). If the hardware interface does not give access to the ready busy pin, then the function must not be defined and the function pointer this->dev_ready is set to NULL.

Init function

The init function allocates memory and sets up all the board specific parameters and function pointers. When everything is set up nand_scan() is called. This function tries to detect and identify then chip. If a chip is found all the internal data fields are initialized accordingly. The structure(s) have to be zeroed out first and then filled with the necessary information about the device.

```
static int __init board_init (void)
{
    struct nand_chip *this;
    int err = 0;

    /* Allocate memory for MTD device structure and private data */
    this = kzalloc(sizeof(struct nand_chip), GFP_KERNEL);
    if (!this) {
        printk ("Unable to allocate NAND MTD device structure.\n");
        err = -ENOMEM;
        goto out;
    }

    board_mtd = nand_to_mtd(this);

    /* map physical address */
    baseaddr = ioremap(CHIP_PHYSICAL_ADDRESS, 1024);
    if (!baseaddr) {
        printk("Ioremap to access NAND chip failed\n");
        err = -EIO;
        goto out_mtd;
    }

    /* Set address of NAND IO lines */
    this->IO_ADDR_R = baseaddr;
    this->IO_ADDR_W = baseaddr;
    /* Reference hardware control function */
    this->hwcontrol = board_hwcontrol;
    /* Set command delay time, see datasheet for correct value */
    this->chip_delay = CHIP_DEPENDEND_COMMAND_DELAY;
    /* Assign the device ready function, if available */
    this->dev_ready = board_dev_ready;
    this->eccmode = NAND_ECC_SOFT;

    /* Scan to find existence of the device */
    if (nand_scan (board_mtd, 1)) {
        err = -ENXIO;
        goto out_ior;
    }

    add_mtd_partitions(board_mtd, partition_info, NUM_PARTITIONS);
    goto out;

out_ior:
    iounmap(baseaddr);
out_mtd:
    kfree (this);
out:
    return err;
}
module_init(board_init);
```

Exit function

The exit function is only necessary if the driver is compiled as a module. It releases all resources which are held by the chip driver and unregisters the partitions in the MTD layer.

```
#ifdef MODULE
static void __exit board_cleanup (void)
{
    /* Release resources, unregister device */
    nand_release (board_mtd);

    /* unmap physical address */
    iounmap(baseaddr);

    /* Free the MTD device structure */
    kfree (mtd_to_nand(board_mtd));
}
module_exit(board_cleanup);
#endif
```

Advanced board driver functions

This chapter describes the advanced functionality of the NAND driver. For a list of functions which can be overridden by the board driver see the documentation of the nand_chip structure.

Multiple chip control

The nand driver can control chip arrays. Therefore the board driver must provide an own select_chip function. This function must (de)select the requested chip. The function pointer in the nand_chip structure must be set before calling nand_scan(). The maxchip parameter of nand_scan() defines the maximum number of chips to scan for. Make sure that the select_chip function can handle the requested number of chips.

The nand driver concatenates the chips to one virtual chip and provides this virtual chip to the MTD layer.

Note: The driver can only handle linear chip arrays of equally sized chips. There is no support for parallel arrays which extend the buswidth.

GPIO based example

```
static void board_select_chip (struct mtd_info *mtd, int chip)
{
    /* Deselect all chips, set all nCE pins high */
    GPIO(BOARD_NAND_NCE) |= 0xff;
    if (chip >= 0)
        GPIO(BOARD_NAND_NCE) &= ~ (1 << chip);
}
```

Address lines based example. Its assumed that the nCE pins are connected to an address decoder.

```
static void board_select_chip (struct mtd_info *mtd, int chip)
{
    struct nand_chip *this = mtd_to_nand(mtd);

    /* Deselect all chips */
    this->IO_ADDR_R &= ~BOARD_NAND_ADDR_MASK;
    this->IO_ADDR_W &= ~BOARD_NAND_ADDR_MASK;
    switch (chip) {
    case 0:
        this->IO_ADDR_R |= BOARD_NAND_ADDR_CHIP0;
        this->IO_ADDR_W |= BOARD_NAND_ADDR_CHIP0;
        break;
    ....
    case n:
        this->IO_ADDR_R |= BOARD_NAND_ADDR_CHIPn;
        this->IO_ADDR_W |= BOARD_NAND_ADDR_CHIPn;
        break;
    }
}
```

Hardware ECC support

Functions and constants

The nand driver supports three different types of hardware ECC.

- NAND_ECC_HW3_256

  Hardware ECC generator providing 3 bytes ECC per 256 byte.

- NAND_ECC_HW3_512

  Hardware ECC generator providing 3 bytes ECC per 512 byte.

- NAND_ECC_HW6_512

  Hardware ECC generator providing 6 bytes ECC per 512 byte.

- NAND_ECC_HW8_512

  Hardware ECC generator providing 6 bytes ECC per 512 byte.

If your hardware generator has a different functionality add it at the appropriate place in nand_base.c

The board driver must provide following functions:

- enable_hwecc

  This function is called before reading / writing to the chip. Reset or initialize the hardware generator in this function. The function is called with an argument which let you distinguish between read and write operations.

- calculate_ecc

  This function is called after read / write from / to the chip. Transfer the ECC from the hardware to the buffer. If the option NAND_HWECC_SYNDROME is set then the function is only called on write. See below.

- correct_data

  In case of an ECC error this function is called for error detection and correction. Return 1 respectively 2 in case the error can be corrected. If the error is not correctable return -1. If your hardware generator matches the default algorithm of the nand_ecc software generator then use the correction function provided by nand_ecc instead of implementing duplicated code.

Hardware ECC with syndrome calculation

Many hardware ECC implementations provide Reed-Solomon codes and calculate an error syndrome on read. The syndrome must be converted to a standard Reed-Solomon syndrome before calling the error correction code in the generic Reed-Solomon library.

The ECC bytes must be placed immediately after the data bytes in order to make the syndrome generator work. This is contrary to the usual layout used by software ECC. The separation of data and out of band area is not longer possible. The nand driver code handles this layout and the remaining free bytes in the oob area are managed by the autoplacement code. Provide a matching oob-layout in this case. See rts_from4.c and diskonchip.c for implementation reference. In those cases we must also use bad block tables on FLASH, because the ECC layout is interfering with the bad block marker positions. See bad block table support for details.

Bad block table support

Most NAND chips mark the bad blocks at a defined position in the spare area. Those blocks must not be erased under any circumstances as the bad block information would be lost. It is possible to check the bad block mark each time when the blocks are accessed by reading the spare area of the first page in the block. This is time consuming so a bad block table is used.

The nand driver supports various types of bad block tables.

- Per device

  The bad block table contains all bad block information of the device which can consist of multiple chips.

- Per chip

  A bad block table is used per chip and contains the bad block information for this particular chip.

- Fixed offset

  The bad block table is located at a fixed offset in the chip (device). This applies to various DiskOnChip devices.

- Automatic placed

  The bad block table is automatically placed and detected either at the end or at the beginning of a chip (device)

- Mirrored tables

  The bad block table is mirrored on the chip (device) to allow updates of the bad block table without data loss.

nand_scan() calls the function nand_default_bbt(). nand_default_bbt() selects appropriate default bad block table descriptors depending on the chip information which was retrieved by nand_scan().

The standard policy is scanning the device for bad blocks and build a ram based bad block table which allows faster access than always checking the bad block information on the flash chip itself.

Flash based tables

It may be desired or necessary to keep a bad block table in FLASH. For AG-AND chips this is mandatory, as they have no factory marked bad blocks. They have factory marked good blocks. The marker pattern is erased when the block is erased to be reused. So in case of powerloss before writing the pattern back to the chip this block would be lost and added to the bad blocks. Therefore we scan the chip(s) when we detect them the first time for good blocks and store this information in a bad block table before erasing any of the blocks.

The blocks in which the tables are stored are protected against accidental access by marking them bad in the memory bad block table. The bad block table management functions are allowed to circumvent this protection.

The simplest way to activate the FLASH based bad block table support is to set the option NAND_BBT_USE_FLASH in the bbt_option field of the nand chip structure before calling nand_scan(). For AG-AND chips is this done by default. This activates the default FLASH based bad block table functionality of the NAND driver. The default bad block table options are

- Store bad block table per chip
- Use 2 bits per block
- Automatic placement at the end of the chip
- Use mirrored tables with version numbers
- Reserve 4 blocks at the end of the chip

User defined tables

User defined tables are created by filling out a nand_bbt_descr structure and storing the pointer in the nand_chip structure member bbt_td before calling nand_scan(). If a mirror table is necessary a second structure must be created and a pointer to this structure must be stored in bbt_md inside the nand_chip structure. If the bbt_md member is set to NULL then only the main table is used and no scan for the mirrored table is performed.

The most important field in the nand_bbt_descr structure is the options field. The options define most of the table properties. Use the predefined constants from rawnand.h to define the options.

- Number of bits per block

  The supported number of bits is 1, 2, 4, 8.

- Table per chip

  Setting the constant NAND_BBT_PERCHIP selects that a bad block table is managed for each chip in a chip array. If this option is not set then a per device bad block table is used.

- Table location is absolute

  Use the option constant NAND_BBT_ABSPAGE and define the absolute page number where the bad block table starts in the field pages. If you have selected bad block tables per chip and you have a multi chip array then the start page must be given for each chip in the chip array. Note: there is no scan for a table ident pattern performed, so the fields pattern, veroffs, offs, len can be left uninitialized

- Table location is automatically detected

  The table can either be located in the first or the last good blocks of the chip (device). Set NAND_BBT_LASTBLOCK to place the bad block table at the end of the chip (device). The bad block tables are marked and identified by a pattern which is stored in the spare area of the first page in the block which holds the bad block table. Store a pointer to the pattern in the pattern field. Further the length of the pattern has to be stored in len and the offset in the spare area must be given in the offs member of the nand_bbt_descr structure. For mirrored bad block tables different patterns are mandatory.

- Table creation

  Set the option NAND_BBT_CREATE to enable the table creation if no table can be found during the scan. Usually this is done only once if a new chip is found.

- Table write support

  Set the option NAND_BBT_WRITE to enable the table write support. This allows the update of the bad block table(s) in case a block has to be marked bad due to wear. The MTD interface function block_markbad is calling the update function of the bad block table. If the write support is enabled then the table is updated on FLASH.

  Note: Write support should only be enabled for mirrored tables with version control.

- Table version control

  Set the option NAND_BBT_VERSION to enable the table version control. It's highly recommended to enable this for mirrored tables with write support. It makes sure that the risk of losing the bad block table information is reduced to the loss of the information about the one worn out block which should be marked bad. The version is stored in 4 consecutive bytes in the spare area of the device. The position of the version number is defined by the member veroffs in the bad block table descriptor.

- Save block contents on write

  In case that the block which holds the bad block table does contain other useful information, set the option NAND_BBT_SAVECONTENT. When the bad block table is written then the whole block is read the bad block table is updated and the block is erased and everything is written back. If this option is not set only the bad block table is written and everything else in the block is ignored and erased.

- Number of reserved blocks

  For automatic placement some blocks must be reserved for bad block table storage. The number of reserved blocks is defined in the maxblocks member of the bad block table description structure. Reserving 4 blocks for mirrored tables should be a reasonable number. This also limits the number of blocks which are scanned for the bad block table ident pattern.

Spare area (auto)placement

The nand driver implements different possibilities for placement of filesystem data in the spare area,

- Placement defined by fs driver
- Automatic placement

The default placement function is automatic placement. The nand driver has built in default placement schemes for the various chiptypes. If due to hardware ECC functionality the default placement does not fit then the board driver can provide a own placement scheme.

File system drivers can provide a own placement scheme which is used instead of the default placement scheme.

Placement schemes are defined by a nand_oobinfo structure

```
struct nand_oobinfo {
    int useecc;
    int eccbytes;
    int eccpos[24];
    int oobfree[8][2];
};
```

- useecc

  The useecc member controls the ecc and placement function. The header file include/mtd/mtd-abi.h contains constants to select ecc and placement. MTD_NANDECC_OFF switches off the ecc complete. This is not recommended and available for testing and diagnosis only. MTD_NANDECC_PLACE selects caller defined placement, MTD_NANDECC_AUTOPLACE selects automatic placement.

- eccbytes

  The eccbytes member defines the number of ecc bytes per page.

- eccpos

  The eccpos array holds the byte offsets in the spare area where the ecc codes are placed.

- oobfree

  The oobfree array defines the areas in the spare area which can be used for automatic placement. The information is given in the format {offset, size}. offset defines the start of the usable area, size the length in bytes. More than one area can be defined. The list is terminated by an {0, 0} entry.

Placement defined by fs driver

The calling function provides a pointer to a nand_oobinfo structure which defines the ecc placement. For writes the caller must provide a spare area buffer along with the data buffer. The spare area buffer size is (number of pages) * (size of spare area). For reads the buffer size is (number of pages) * ((size of spare area) + (number of ecc steps per page) * sizeof (int)). The driver stores the result of the ecc check for each tuple in the spare buffer. The storage sequence is:

```
<spare data page 0><ecc result 0>...<ecc result n>

...

<spare data page n><ecc result 0>...<ecc result n>
```

This is a legacy mode used by YAFFS1.

If the spare area buffer is NULL then only the ECC placement is done according to the given scheme in the nand_oobinfo structure.

Automatic placement

Automatic placement uses the built in defaults to place the ecc bytes in the spare area. If filesystem data have to be stored / read into the spare area then the calling function must provide a buffer. The buffer size per page is determined by the oobfree array in the nand_oobinfo structure.

If the spare area buffer is NULL then only the ECC placement is done according to the default builtin scheme.

Spare area autoplacement default schemes

256 byte pagesize

Offset	Content	Comment
0x00	ECC byte 0	Error correction code byte 0
0x01	ECC byte 1	Error correction code byte 1
0x02	ECC byte 2	Error correction code byte 2
0x03	Autoplace 0	
0x04	Autoplace 1	
0x05	Bad block marker	If any bit in this byte is zero, then this block is bad. This applies only to the first page in a block. In the remaining pages this byte is reserved
0x06	Autoplace 2	
0x07	Autoplace 3	

512 byte pagesize

Offset	Content	Comment
0x00	ECC byte 0	Error correction code byte 0 of the lower 256 Byte data in this page
0x01	ECC byte 1	Error correction code byte 1 of the lower 256 Bytes of data in this page
0x02	ECC byte 2	Error correction code byte 2 of the lower 256 Bytes of data in this page
0x03	ECC byte 3	Error correction code byte 0 of the upper 256 Bytes of data in this page
0x04	reserved	reserved
0x05	Bad block marker	If any bit in this byte is zero, then this block is bad. This applies only to the first page in a block. In the remaining pages this byte is reserved
0x06	ECC byte 4	Error correction code byte 1 of the upper 256 Bytes of data in this page
0x07	ECC byte 5	Error correction code byte 2 of the upper 256 Bytes of data in this page
0x08 - 0x0F	Autoplace 0 - 7	

2048 byte pagesize

Offset	Content	Comment
0x00	Bad block marker	If any bit in this byte is zero, then this block is bad. This applies only to the first page in a block. In the remaining pages this byte is reserved
0x01	Reserved	Reserved
0x02-0x27	Autoplace 0 - 37	
0x28	ECC byte 0	Error correction code byte 0 of the first 256 Byte data in this page
0x29	ECC byte 1	Error correction code byte 1 of the first 256 Bytes of data in this page
0x2A	ECC byte 2	Error correction code byte 2 of the first 256 Bytes data in this page
0x2B	ECC byte 3	Error correction code byte 0 of the second 256 Bytes of data in this page
0x2C	ECC byte 4	Error correction code byte 1 of the second 256 Bytes of data in this page
0x2D	ECC byte 5	Error correction code byte 2 of the second 256 Bytes of data in this page
0x2E	ECC byte 6	Error correction code byte 0 of the third 256 Bytes of data in this page
0x2F	ECC byte 7	Error correction code byte 1 of the third 256 Bytes of data in this page
0x30	ECC byte 8	Error correction code byte 2 of the third 256 Bytes of data in this page
0x31	ECC byte 9	Error correction code byte 0 of the fourth 256 Bytes of data in this page
0x32	ECC byte 10	Error correction code byte 1 of the fourth 256 Bytes of data in this page
0x33	ECC byte 11	Error correction code byte 2 of the fourth 256 Bytes of data in this page
0x34	ECC byte 12	Error correction code byte 0 of the fifth 256 Bytes of data in this page
0x35	ECC byte 13	Error correction code byte 1 of the fifth 256 Bytes of data in this page
0x36	ECC byte 14	Error correction code byte 2 of the fifth 256 Bytes of data in this page
0x37	ECC byte 15	Error correction code byte 0 of the sixth 256 Bytes of data in this page
0x38	ECC byte 16	Error correction code byte 1 of the sixth 256 Bytes of data in this page
0x39	ECC byte 17	Error correction code byte 2 of the sixth 256 Bytes of data in this page
0x3A	ECC byte 18	Error correction code byte 0 of the seventh 256 Bytes of data in this page
0x3B	ECC byte 19	Error correction code byte 1 of the seventh 256 Bytes of data in this page
0x3C	ECC byte 20	Error correction code byte 2 of the seventh 256 Bytes of data in this page
0x3D	ECC byte 21	Error correction code byte 0 of the eighth 256 Bytes of data in this page
0x3E	ECC byte 22	Error correction code byte 1 of the eighth 256 Bytes of data in this page
0x3F	ECC byte 23	Error correction code byte 2 of the eighth 256 Bytes of data in this page

Filesystem support

The NAND driver provides all necessary functions for a filesystem via the MTD interface.

Filesystems must be aware of the NAND peculiarities and restrictions. One major restrictions of NAND Flash is, that you cannot write as often as you want to a page. The consecutive writes to a page, before erasing it again, are restricted to 1-3 writes, depending on the manufacturers specifications. This applies similar to the spare area.

Therefore NAND aware filesystems must either write in page size chunks or hold a writebuffer to collect smaller writes until they sum up to pagesize. Available NAND aware filesystems: JFFS2, YAFFS.

The spare area usage to store filesystem data is controlled by the spare area placement functionality which is described in one of the earlier chapters.

Tools

The MTD project provides a couple of helpful tools to handle NAND Flash.

- flasherase, flasheraseall: Erase and format FLASH partitions
- nandwrite: write filesystem images to NAND FLASH
- nanddump: dump the contents of a NAND FLASH partitions

These tools are aware of the NAND restrictions. Please use those tools instead of complaining about errors which are caused by non NAND aware access methods.

Constants

This chapter describes the constants which might be relevant for a driver developer.

Chip option constants

Constants for chip id table

These constants are defined in rawnand.h. They are OR-ed together to describe the chip functionality:

```
/* Buswitdh is 16 bit */
#define NAND_BUSWIDTH_16    0x00000002
/* Device supports partial programming without padding */
#define NAND_NO_PADDING     0x00000004
/* Chip has cache program function */
#define NAND_CACHEPRG       0x00000008
/* Chip has copy back function */
#define NAND_COPYBACK       0x00000010
/* AND Chip which has 4 banks and a confusing page / block
 * assignment. See Renesas datasheet for further information */
#define NAND_IS_AND     0x00000020
/* Chip has a array of 4 pages which can be read without
 * additional ready /busy waits */
#define NAND_4PAGE_ARRAY    0x00000040
```

Constants for runtime options

These constants are defined in rawnand.h. They are OR-ed together to describe the functionality:

```
/* The hw ecc generator provides a syndrome instead a ecc value on read
 * This can only work if we have the ecc bytes directly behind the
 * data bytes. Applies for DOC and AG-AND Renesas HW Reed Solomon generators */
#define NAND_HWECC_SYNDROME 0x00020000
```

ECC selection constants

Use these constants to select the ECC algorithm:

```
/* No ECC. Usage is not recommended ! */
#define NAND_ECC_NONE       0
/* Software ECC 3 byte ECC per 256 Byte data */
#define NAND_ECC_SOFT       1
/* Hardware ECC 3 byte ECC per 256 Byte data */
#define NAND_ECC_HW3_256    2
/* Hardware ECC 3 byte ECC per 512 Byte data */
#define NAND_ECC_HW3_512    3
/* Hardware ECC 6 byte ECC per 512 Byte data */
#define NAND_ECC_HW6_512    4
/* Hardware ECC 6 byte ECC per 512 Byte data */
#define NAND_ECC_HW8_512    6
```

Hardware control related constants

These constants describe the requested hardware access function when the boardspecific hardware control function is called:

```
/* Select the chip by setting nCE to low */
#define NAND_CTL_SETNCE     1
/* Deselect the chip by setting nCE to high */
#define NAND_CTL_CLRNCE     2
/* Select the command latch by setting CLE to high */
#define NAND_CTL_SETCLE     3
/* Deselect the command latch by setting CLE to low */
#define NAND_CTL_CLRCLE     4
/* Select the address latch by setting ALE to high */
#define NAND_CTL_SETALE     5
/* Deselect the address latch by setting ALE to low */
#define NAND_CTL_CLRALE     6
/* Set write protection by setting WP to high. Not used! */
#define NAND_CTL_SETWP      7
/* Clear write protection by setting WP to low. Not used! */
#define NAND_CTL_CLRWP      8
```

Bad block table related constants

These constants describe the options used for bad block table descriptors:

```
/* Options for the bad block table descriptors */

/* The number of bits used per block in the bbt on the device */
#define NAND_BBT_NRBITS_MSK 0x0000000F
#define NAND_BBT_1BIT       0x00000001
#define NAND_BBT_2BIT       0x00000002
#define NAND_BBT_4BIT       0x00000004
#define NAND_BBT_8BIT       0x00000008
/* The bad block table is in the last good block of the device */
#define NAND_BBT_LASTBLOCK  0x00000010
```

```
/* The bbt is at the given page, else we must scan for the bbt */
#define NAND_BBT_ABSPAGE    0x00000020
/* bbt is stored per chip on multichip devices */
#define NAND_BBT_PERCHIP    0x00000080
/* bbt has a version counter at offset veroffs */
#define NAND_BBT_VERSION    0x00000100
/* Create a bbt if none axists */
#define NAND_BBT_CREATE     0x00000200
/* Write bbt if necessary */
#define NAND_BBT_WRITE      0x00001000
/* Read and write back block contents when writing bbt */
#define NAND_BBT_SAVECONTENT    0x00002000
```

Structures

This chapter contains the autogenerated documentation of the structures which are used in the NAND driver and might be relevant for a driver developer. Each struct member has a short description which is marked with an [XXX] identifier. See the chapter “Documentation hints” for an explanation.

struct **nand_id**
NAND id structure

Definition

```
struct nand_id {
  u8 data[NAND_MAX_ID_LEN];
  int len;
};
```

Members

data buffer containing the id bytes.

len ID length.

struct **nand_hw_control**
Control structure for hardware controller (e.g ECC generator) shared among independent devices

Definition

```
struct nand_hw_control {
  spinlock_t lock;
  struct nand_chip *active;
  wait_queue_head_t wq;
};
```

Members

lock protection lock

active the mtd device which holds the controller currently

wq wait queue to sleep on if a NAND operation is in progress used instead of the per chip wait queue when a hw controller is available.

struct **nand_ecc_step_info**
ECC step information of ECC engine

Definition

```
struct nand_ecc_step_info {
  int stepsize;
  const int *strengths;
  int nstrengths;
};
```

Members

stepsize data bytes per ECC step

strengths array of supported strengths

nstrengths number of supported strengths

struct **nand_ecc_caps**
capability of ECC engine

Definition

```
struct nand_ecc_caps {
  const struct nand_ecc_step_info *stepinfos;
  int nstepinfos;
  int (*calc_ecc_bytes)(int step_size, int strength);
};
```

Members

stepinfos array of ECC step information

nstepinfos number of ECC step information

calc_ecc_bytes driver's hook to calculate ECC bytes per step

struct **nand_ecc_ctrl**
Control structure for ECC

Definition

```
struct nand_ecc_ctrl {
  nand_ecc_modes_t mode;
  enum nand_ecc_algo algo;
  int steps;
  int size;
  int bytes;
  int total;
  int strength;
  int prepad;
  int postpad;
  unsigned int options;
  void *priv;
  u8 *calc_buf;
  u8 *code_buf;
  void (*hwctl)(struct mtd_info *mtd, int mode);
  int (*calculate)(struct mtd_info *mtd, const uint8_t *dat, uint8_t *ecc_code);
  int (*correct)(struct mtd_info *mtd, uint8_t *dat, uint8_t *read_ecc, uint8_t *calc_ecc);
  int (*read_page_raw)(struct mtd_info *mtd, struct nand_chip *chip, uint8_t *buf, int oob_required, int
  int (*write_page_raw)(struct mtd_info *mtd, struct nand_chip *chip, const uint8_t *buf, int oob_requir
  int (*read_page)(struct mtd_info *mtd, struct nand_chip *chip, uint8_t *buf, int oob_required, int pag
  int (*read_subpage)(struct mtd_info *mtd, struct nand_chip *chip, uint32_t offs, uint32_t len, uint8_t
  int (*write_subpage)(struct mtd_info *mtd, struct nand_chip *chip,uint32_t offset, uint32_t data_len,
  int (*write_page)(struct mtd_info *mtd, struct nand_chip *chip, const uint8_t *buf, int oob_required,
  int (*write_oob_raw)(struct mtd_info *mtd, struct nand_chip *chip, int page);
  int (*read_oob_raw)(struct mtd_info *mtd, struct nand_chip *chip, int page);
  int (*read_oob)(struct mtd_info *mtd, struct nand_chip *chip, int page);
  int (*write_oob)(struct mtd_info *mtd, struct nand_chip *chip, int page);
};
```

Members

mode ECC mode

algo ECC algorithm

steps number of ECC steps per page

size data bytes per ECC step

bytes ECC bytes per step

total total number of ECC bytes per page

strength max number of correctible bits per ECC step

prepad padding information for syndrome based ECC generators

postpad padding information for syndrome based ECC generators

options ECC specific options (see NAND_ECC_XXX flags defined above)

priv pointer to private ECC control data

calc_buf buffer for calculated ECC, size is oobsize.

code_buf buffer for ECC read from flash, size is oobsize.

hwctl function to control hardware ECC generator. Must only be provided if an hardware ECC is available

calculate function for ECC calculation or readback from ECC hardware

correct function for ECC correction, matching to ECC generator (sw/hw). Should return a positive number representing the number of corrected bitflips, -EBADMSG if the number of bitflips exceed ECC strength, or any other error code if the error is not directly related to correction. If -EBADMSG is returned the input buffers should be left untouched.

read_page_raw function to read a raw page without ECC. This function should hide the specific layout used by the ECC controller and always return contiguous in-band and out-of-band data even if they're not stored contiguously on the NAND chip (e.g. NAND_ECC_HW_SYNDROME interleaves in-band and out-of-band data).

write_page_raw function to write a raw page without ECC. This function should hide the specific layout used by the ECC controller and consider the passed data as contiguous in-band and out-of-band data. ECC controller is responsible for doing the appropriate transformations to adapt to its specific layout (e.g. NAND_ECC_HW_SYNDROME interleaves in-band and out-of-band data).

read_page function to read a page according to the ECC generator requirements; returns maximum number of bitflips corrected in any single ECC step, -EIO hw error

read_subpage function to read parts of the page covered by ECC; returns same as read_page()

write_subpage function to write parts of the page covered by ECC.

write_page function to write a page according to the ECC generator requirements.

write_oob_raw function to write chip OOB data without ECC

read_oob_raw function to read chip OOB data without ECC

read_oob function to read chip OOB data

write_oob function to write chip OOB data

struct **nand_sdr_timings**
SDR NAND chip timings

Definition

```
struct nand_sdr_timings {
  u64 tBERS_max;
  u32 tCCS_min;
  u64 tPROG_max;
  u64 tR_max;
  u32 tALH_min;
  u32 tADL_min;
  u32 tALS_min;
  u32 tAR_min;
  u32 tCEA_max;
  u32 tCEH_min;
  u32 tCH_min;
  u32 tCHZ_max;
  u32 tCLH_min;
  u32 tCLR_min;
  u32 tCLS_min;
  u32 tCOH_min;
  u32 tCS_min;
  u32 tDH_min;
  u32 tDS_min;
  u32 tFEAT_max;
  u32 tIR_min;
  u32 tITC_max;
  u32 tRC_min;
  u32 tREA_max;
  u32 tREH_min;
  u32 tRHOH_min;
  u32 tRHW_min;
  u32 tRHZ_max;
  u32 tRLOH_min;
  u32 tRP_min;
  u32 tRR_min;
  u64 tRST_max;
  u32 tWB_max;
  u32 tWC_min;
  u32 tWH_min;
  u32 tWHR_min;
  u32 tWP_min;
  u32 tWW_min;
};
```

Members

tBERS_max Block erase time

tCCS_min Change column setup time

tPROG_max Page program time

tR_max Page read time

tALH_min ALE hold time

tADL_min ALE to data loading time

tALS_min ALE setup time

tAR_min ALE to RE# delay

tCEA_max CE# access time

tCEH_min CE# high hold time

tCH_min CE# hold time

tCHZ_max CE# high to output hi-Z

tCLH_min CLE hold time

tCLR_min CLE to RE# delay

tCLS_min CLE setup time

tCOH_min CE# high to output hold

tCS_min CE# setup time

tDH_min Data hold time

tDS_min Data setup time

tFEAT_max Busy time for Set Features and Get Features

tIR_min Output hi-Z to RE# low

tITC_max Interface and Timing Mode Change time

tRC_min RE# cycle time

tREA_max RE# access time

tREH_min RE# high hold time

tRHOH_min RE# high to output hold

tRHW_min RE# high to WE# low

tRHZ_max RE# high to output hi-Z

tRLOH_min RE# low to output hold

tRP_min RE# pulse width

tRR_min Ready to RE# low (data only)

tRST_max Device reset time, measured from the falling edge of R/B# to the rising edge of R/B#.

tWB_max WE# high to SR[6] low

tWC_min WE# cycle time

tWH_min WE# high hold time

tWHR_min WE# high to RE# low

tWP_min WE# pulse width

tWW_min WP# transition to WE# low

Description

This struct defines the timing requirements of a SDR NAND chip. These information can be found in every NAND datasheets and the timings meaning are described in the ONFI specifications: www.onfi.org/~/media/ONFI/specs/onfi_3_1_spec.pdf (chapter 4.15 Timing Parameters)

All these timings are expressed in picoseconds.

enum **nand_data_interface_type**
NAND interface timing type

Constants

NAND_SDR_IFACE Single Data Rate interface

struct **nand_data_interface**
NAND interface timing

Definition

```
struct nand_data_interface {
  enum nand_data_interface_type type;
  union {
    struct nand_sdr_timings sdr;
  } timings;
};
```

Members

type type of the timing

timings The timing, type according to **type**

const struct *nand_sdr_timings* * **nand_get_sdr_timings**(const struct *nand_data_interface* * *conf*)
get SDR timing from data interface

Parameters

const struct nand_data_interface * conf The data interface

struct **nand_manufacturer_ops**
NAND Manufacturer operations

Definition

```
struct nand_manufacturer_ops {
  void (*detect)(struct nand_chip *chip);
  int (*init)(struct nand_chip *chip);
  void (*cleanup)(struct nand_chip *chip);
};
```

Members

detect detect the NAND memory organization and capabilities

init initialize all vendor specific fields (like the ->:c:func:*read_retry()* implementation) if any.

cleanup the ->:c:func:*init()* function may have allocated resources, ->:c:func:*cleanup()* is here to let vendor specific code release those resources.

struct **nand_op_cmd_instr**
Definition of a command instruction

Definition

```
struct nand_op_cmd_instr {
  u8 opcode;
};
```

Members

opcode the command to issue in one cycle

struct **nand_op_addr_instr**
Definition of an address instruction

Definition

```
struct nand_op_addr_instr {
  unsigned int naddrs;
  const u8 *addrs;
};
```

Members

naddrs length of the **addrs** array

addrs array containing the address cycles to issue

struct **nand_op_data_instr**
Definition of a data instruction

Definition

```
struct nand_op_data_instr {
  unsigned int len;
  union {
    void *in;
    const void *out;
  } buf;
  bool force_8bit;
};
```

Members

len number of data bytes to move

force_8bit force 8-bit access

Description

Please note that "in" and "out" are inverted from the ONFI specification and are from the controller perspective, so a "in" is a read from the NAND chip while a "out" is a write to the NAND chip.

struct **nand_op_waitrdy_instr**
Definition of a wait ready instruction

Definition

```
struct nand_op_waitrdy_instr {
  unsigned int timeout_ms;
};
```

Members

timeout_ms maximum delay while waiting for the ready/busy pin in ms

enum **nand_op_instr_type**
Definition of all instruction types

Constants

NAND_OP_CMD_INSTR command instruction

NAND_OP_ADDR_INSTR address instruction

NAND_OP_DATA_IN_INSTR data in instruction

NAND_OP_DATA_OUT_INSTR data out instruction

NAND_OP_WAITRDY_INSTR wait ready instruction

struct **nand_op_instr**
Instruction object

Definition

```
struct nand_op_instr {
  enum nand_op_instr_type type;
  union {
    struct nand_op_cmd_instr cmd;
    struct nand_op_addr_instr addr;
    struct nand_op_data_instr data;
    struct nand_op_waitrdy_instr waitrdy;
  } ctx;
  unsigned int delay_ns;
};
```

Members

type the instruction type **cmd/addr/data/waitrdy**: extra data associated to the instruction.

You'll have to use the appropriate element depending on **type**

delay_ns delay the controller should apply after the instruction has been issued on the bus. Most modern controllers have internal timings control logic, and in this case, the controller driver can ignore this field.

struct **nand_subop**
a sub operation

Definition

```
struct nand_subop {
  const struct nand_op_instr *instrs;
  unsigned int ninstrs;
  unsigned int first_instr_start_off;
  unsigned int last_instr_end_off;
};
```

Members

instrs array of instructions

ninstrs length of the **instrs** array

first_instr_start_off offset to start from for the first instruction of the sub-operation

last_instr_end_off offset to end at (excluded) for the last instruction of the sub-operation

Description

Both **first_instr_start_off** and **last_instr_end_off** only apply to data or address instructions.

When an operation cannot be handled as is by the NAND controller, it will be split by the parser into sub-operations which will be passed to the controller driver.

struct **nand_op_parser_addr_constraints**
Constraints for address instructions

Definition

```
struct nand_op_parser_addr_constraints {
  unsigned int maxcycles;
};
```

Members

maxcycles maximum number of address cycles the controller can issue in a single step

struct **nand_op_parser_data_constraints**
Constraints for data instructions

Definition

```
struct nand_op_parser_data_constraints {
  unsigned int maxlen;
};
```

Members

maxlen maximum data length that the controller can handle in a single step

struct **nand_op_parser_pattern_elem**
One element of a pattern

Definition

```
struct nand_op_parser_pattern_elem {
  enum nand_op_instr_type type;
  bool optional;
  union {
    struct nand_op_parser_addr_constraints addr;
    struct nand_op_parser_data_constraints data;
  } ctx;
};
```

Members

type the instructuction type

optional whether this element of the pattern is optional or mandatory **addr/data**: address or data constraint (number of cycles or data length)

struct **nand_op_parser_pattern**
NAND sub-operation pattern descriptor

Definition

```
struct nand_op_parser_pattern {
  const struct nand_op_parser_pattern_elem *elems;
  unsigned int nelems;
  int (*exec)(struct nand_chip *chip, const struct nand_subop *subop);
};
```

Members

elems array of pattern elements

nelems number of pattern elements in **elems** array

exec the function that will issue a sub-operation

Description

A pattern is a list of elements, each element reprensenting one instruction with its constraints. The pattern itself is used by the core to match NAND chip operation with NAND controller operations. Once a match between a NAND controller operation pattern and a NAND chip operation (or a sub-set of a NAND operation) is found, the pattern ->:c:func:*exec()* hook is called so that the controller driver can issue the operation on the bus.

Controller drivers should declare as many patterns as they support and pass this list of patterns (created with the help of the following macro) to the *nand_op_parser_exec_op()* helper.

struct **nand_op_parser**
NAND controller operation parser descriptor

Definition

```
struct nand_op_parser {
  const struct nand_op_parser_pattern *patterns;
  unsigned int npatterns;
};
```

Members

patterns array of supported patterns

npatterns length of the **patterns** array

Description

The parser descriptor is just an array of supported patterns which will be iterated by *nand_op_parser_exec_op()* everytime it tries to execute an NAND operation (or tries to determine if a specific operation is supported).

It is worth mentioning that patterns will be tested in their declaration order, and the first match will be taken, so it's important to order patterns appropriately so that simple/inefficient patterns are placed at the end of the list. Usually, this is where you put single instruction patterns.

struct **nand_operation**
NAND operation descriptor

Definition

```
struct nand_operation {
  const struct nand_op_instr *instrs;
  unsigned int ninstrs;
};
```

Members

instrs array of instructions to execute

ninstrs length of the **instrs** array

Description

The actual operation structure that will be passed to chip->:c:func:*exec_op()*.

struct **nand_chip**
NAND Private Flash Chip Data

Definition

```
struct nand_chip {
  struct mtd_info mtd;
  void __iomem *IO_ADDR_R;
  void __iomem *IO_ADDR_W;
  uint8_t (*read_byte)(struct mtd_info *mtd);
  u16 (*read_word)(struct mtd_info *mtd);
  void (*write_byte)(struct mtd_info *mtd, uint8_t byte);
  void (*write_buf)(struct mtd_info *mtd, const uint8_t *buf, int len);
  void (*read_buf)(struct mtd_info *mtd, uint8_t *buf, int len);
  void (*select_chip)(struct mtd_info *mtd, int chip);
  int (*block_bad)(struct mtd_info *mtd, loff_t ofs);
  int (*block_markbad)(struct mtd_info *mtd, loff_t ofs);
  void (*cmd_ctrl)(struct mtd_info *mtd, int dat, unsigned int ctrl);
  int (*dev_ready)(struct mtd_info *mtd);
  void (*cmdfunc)(struct mtd_info *mtd, unsigned command, int column, int page_addr);
  int(*waitfunc)(struct mtd_info *mtd, struct nand_chip *this);
  int (*exec_op)(struct nand_chip *chip,const struct nand_operation *op, bool check_only);
  int (*erase)(struct mtd_info *mtd, int page);
  int (*scan_bbt)(struct mtd_info *mtd);
  int (*onfi_set_features)(struct mtd_info *mtd, struct nand_chip *chip, int feature_addr, uint8_t *subf
  int (*onfi_get_features)(struct mtd_info *mtd, struct nand_chip *chip, int feature_addr, uint8_t *subf
  int (*setup_read_retry)(struct mtd_info *mtd, int retry_mode);
  int (*setup_data_interface)(struct mtd_info *mtd, int chipnr, const struct nand_data_interface *conf);
  int chip_delay;
  unsigned int options;
  unsigned int bbt_options;
  int page_shift;
  int phys_erase_shift;
  int bbt_erase_shift;
  int chip_shift;
  int numchips;
  uint64_t chipsize;
  int pagemask;
  u8 *data_buf;
  int pagebuf;
  unsigned int pagebuf_bitflips;
  int subpagesize;
  uint8_t bits_per_cell;
  uint16_t ecc_strength_ds;
  uint16_t ecc_step_ds;
  int onfi_timing_mode_default;
  int badblockpos;
  int badblockbits;
  struct nand_id id;
  int onfi_version;
  int jedec_version;
  union {
    struct nand_onfi_params onfi_params;
    struct nand_jedec_params jedec_params;
  };
  u16 max_bb_per_die;
  u32 blocks_per_die;
  struct nand_data_interface data_interface;
  int read_retries;
  flstate_t state;
  uint8_t *oob_poi;
  struct nand_hw_control *controller;
  struct nand_ecc_ctrl ecc;
  unsigned long buf_align;
  struct nand_hw_control hwcontrol;
  uint8_t *bbt;
  struct nand_bbt_descr *bbt_td;
  struct nand_bbt_descr *bbt_md;
  struct nand_bbt_descr *badblock_pattern;
  void *priv;
  struct {
    const struct nand_manufacturer *desc;
    void *priv;
  } manufacturer;
};
```

Members

`mtd` MTD device registered to the MTD framework

`IO_ADDR_R` [BOARDSPECIFIC] address to read the 8 I/O lines of the flash device

`IO_ADDR_W` [BOARDSPECIFIC] address to write the 8 I/O lines of the flash device.

`read_byte` [REPLACEABLE] read one byte from the chip

`read_word` [REPLACEABLE] read one word from the chip

`write_byte` [REPLACEABLE] write a single byte to the chip on the low 8 I/O lines

`write_buf` [REPLACEABLE] write data from the buffer to the chip

`read_buf` [REPLACEABLE] read data from the chip into the buffer

`select_chip` [REPLACEABLE] select chip nr

`block_bad` [REPLACEABLE] check if a block is bad, using OOB markers

`block_markbad` [REPLACEABLE] mark a block bad

`cmd_ctrl` [BOARDSPECIFIC] hardwarespecific function for controlling ALE/CLE/nCE. Also used to write command and address

`dev_ready` [BOARDSPECIFIC] hardwarespecific function for accessing device ready/busy line. If set to NULL no access to ready/busy is available and the ready/busy information is read from the chip status register.

`cmdfunc` [REPLACEABLE] hardwarespecific function for writing commands to the chip.

`waitfunc` [REPLACEABLE] hardwarespecific function for wait on ready.

`exec_op` controller specific method to execute NAND operations. This method replaces ->:c:func:*cmdfunc()*, ->{read,write}_{buf,byte,word}(), ->:c:func:*dev_ready()* and ->:c:func:*waifunc()*.

`erase` [REPLACEABLE] erase function

`scan_bbt` [REPLACEABLE] function to scan bad block table

`onfi_set_features` [REPLACEABLE] set the features for ONFI nand

`onfi_get_features` [REPLACEABLE] get the features for ONFI nand

`setup_read_retry` [FLASHSPECIFIC] flash (vendor) specific function for setting the read-retry mode. Mostly needed for MLC NAND.

setup_data_interface [OPTIONAL] setup the data interface and timing. If chipnr is set to NAND_DATA_IFACE_CHECK_ONLY this means the configuration should not be applied but only checked.

chip_delay [BOARDSPECIFIC] chip dependent delay for transferring data from array to read regs (tR).

options [BOARDSPECIFIC] various chip options. They can partly be set to inform nand_scan about special functionality. See the defines for further explanation.

bbt_options [INTERN] bad block specific options. All options used here must come from bbm.h. By default, these options will be copied to the appropriate nand_bbt_descr's.

page_shift [INTERN] number of address bits in a page (column address bits).

phys_erase_shift [INTERN] number of address bits in a physical eraseblock

bbt_erase_shift [INTERN] number of address bits in a bbt entry

chip_shift [INTERN] number of address bits in one chip

numchips [INTERN] number of physical chips

chipsize [INTERN] the size of one chip for multichip arrays

pagemask [INTERN] page number mask = number of (pages / chip) - 1

data_buf [INTERN] buffer for data, size is (page size + oobsize).

pagebuf [INTERN] holds the pagenumber which is currently in data_buf.

pagebuf_bitflips [INTERN] holds the bitflip count for the page which is currently in data_buf.

subpagesize [INTERN] holds the subpagesize

bits_per_cell [INTERN] number of bits per cell. i.e., 1 means SLC.

ecc_strength_ds [INTERN] ECC correctability from the datasheet. Minimum amount of bit errors per **ecc_step_ds** guaranteed to be correctable. If unknown, set to zero.

ecc_step_ds [INTERN] ECC step required by the **ecc_strength_ds**, also from the datasheet. It is the recommended ECC step size, if known; if unknown, set to zero.

onfi_timing_mode_default [INTERN] default ONFI timing mode. This field is set to the actually used ONFI mode if the chip is ONFI compliant or deduced from the datasheet if the NAND chip is not ONFI compliant.

badblockpos [INTERN] position of the bad block marker in the oob area.

badblockbits [INTERN] minimum number of set bits in a good block's bad block marker position; i.e., BBM == 11110111b is not bad when badblockbits == 7

id [INTERN] holds NAND ID

onfi_version [INTERN] holds the chip ONFI version (BCD encoded), non 0 if ONFI supported.

jedec_version [INTERN] holds the chip JEDEC version (BCD encoded), non 0 if JEDEC supported.

{unnamed_union} anonymous

onfi_params [INTERN] holds the ONFI page parameter when ONFI is supported, 0 otherwise.

jedec_params [INTERN] holds the JEDEC parameter page when JEDEC is supported, 0 otherwise.

max_bb_per_die [INTERN] the max number of bad blocks each die of a this nand device will encounter their life times.

blocks_per_die [INTERN] The number of PEBs in a die

data_interface [INTERN] NAND interface timing information

read_retries [INTERN] the number of read retry modes supported

state [INTERN] the current state of the NAND device

oob_poi "poison value buffer," used for laying out OOB data before writing

controller [REPLACEABLE] a pointer to a hardware controller structure which is shared among multiple independent devices.

ecc [BOARDSPECIFIC] ECC control structure

buf_align minimum buffer alignment required by a platform

hwcontrol platform-specific hardware control structure

bbt [INTERN] bad block table pointer

bbt_td [REPLACEABLE] bad block table descriptor for flash lookup.

bbt_md [REPLACEABLE] bad block table mirror descriptor

badblock_pattern [REPLACEABLE] bad block scan pattern used for initial bad block scan.

priv [OPTIONAL] pointer to private chip data

manufacturer [INTERN] Contains manufacturer information

struct **nand_flash_dev**
NAND Flash Device ID Structure

Definition

```
struct nand_flash_dev {
  char *name;
  union {
    struct {
      uint8_t mfr_id;
      uint8_t dev_id;
    };
    uint8_t id[NAND_MAX_ID_LEN];
  };
  unsigned int pagesize;
  unsigned int chipsize;
  unsigned int erasesize;
  unsigned int options;
  uint16_t id_len;
  uint16_t oobsize;
  struct {
    uint16_t strength_ds;
    uint16_t step_ds;
  } ecc;
  int onfi_timing_mode_default;
};
```

Members

name a human-readable name of the NAND chip

{unnamed_union} anonymous

{unnamed_struct} anonymous

mfr_id manufecturer ID part of the full chip ID array (refers the same memory address as **id**[0])

dev_id device ID part of the full chip ID array (refers the same memory address as **id**[1])

id full device ID array

pagesize size of the NAND page in bytes; if 0, then the real page size (as well as the eraseblock size) is determined from the extended NAND chip ID array)

chipsize total chip size in MiB

erasesize eraseblock size in bytes (determined from the extended ID if 0)

options stores various chip bit options

id_len The valid length of the **id**.

oobsize OOB size

ecc ECC correctability and step information from the datasheet.

ecc.strength_ds The ECC correctability from the datasheet, same as the **ecc_strength_ds** in nand_chip{}.

ecc.step_ds The ECC step required by the **ecc.strength_ds**, same as the **ecc_step_ds** in nand_chip{}, also from the datasheet. For example, the "4bit ECC for each 512Byte" can be set with NAND_ECC_INFO(4, 512).

onfi_timing_mode_default the default ONFI timing mode entered after a NAND reset. Should be deduced from timings described in the datasheet.

struct **nand_manufacturer**
NAND Flash Manufacturer structure

Definition

```
struct nand_manufacturer {
  int id;
  char *name;
  const struct nand_manufacturer_ops *ops;
};
```

Members

id manufacturer ID code of device.

name Manufacturer name

ops manufacturer operations

struct **platform_nand_chip**
chip level device structure

Definition

```
struct platform_nand_chip {
  int nr_chips;
  int chip_offset;
  int nr_partitions;
  struct mtd_partition *partitions;
  int chip_delay;
  unsigned int options;
  unsigned int bbt_options;
  const char **part_probe_types;
};
```

Members

nr_chips max. number of chips to scan for

chip_offset chip number offset

nr_partitions number of partitions pointed to by partitions (or zero)

partitions mtd partition list

chip_delay R/B delay value in us

options Option flags, e.g. 16bit buswidth

bbt_options BBT option flags, e.g. NAND_BBT_USE_FLASH

part_probe_types NULL-terminated array of probe types

struct **platform_nand_ctrl**
controller level device structure

Definition

```
struct platform_nand_ctrl {
  int (*probe)(struct platform_device *pdev);
  void (*remove)(struct platform_device *pdev);
  void (*hwcontrol)(struct mtd_info *mtd, int cmd);
  int (*dev_ready)(struct mtd_info *mtd);
  void (*select_chip)(struct mtd_info *mtd, int chip);
  void (*cmd_ctrl)(struct mtd_info *mtd, int dat, unsigned int ctrl);
  void (*write_buf)(struct mtd_info *mtd, const uint8_t *buf, int len);
  void (*read_buf)(struct mtd_info *mtd, uint8_t *buf, int len);
  unsigned char (*read_byte)(struct mtd_info *mtd);
  void *priv;
};
```

Members

probe platform specific function to probe/setup hardware

remove platform specific function to remove/teardown hardware

hwcontrol platform specific hardware control structure

dev_ready platform specific function to read ready/busy pin

select_chip platform specific chip select function

cmd_ctrl platform specific function for controlling ALE/CLE/nCE. Also used to write command and address

write_buf platform specific function for write buffer

read_buf platform specific function for read buffer

read_byte platform specific function to read one byte from chip

priv private data to transport driver specific settings

Description

All fields are optional and depend on the hardware driver requirements

struct **platform_nand_data**
container structure for platform-specific data

Definition

```
struct platform_nand_data {
  struct platform_nand_chip chip;
  struct platform_nand_ctrl ctrl;
};
```

Members

chip chip level chip structure

ctrl controller level device structure

int **nand_opcode_8bits**(unsigned int *command*)

Parameters

unsigned int command opcode to check

Public Functions Provided

This chapter contains the autogenerated documentation of the NAND kernel API functions which are exported. Each function has a short description which is marked with an [XXX] identifier. See the chapter "Documentation hints" for an explanation.

void **nand_wait_ready**(struct mtd_info * *mtd*)
[GENERIC] Wait for the ready pin after commands.

Parameters

struct mtd_info * mtd MTD device structure

Description

Wait for the ready pin after a command, and warn if a timeout occurs.

int **nand_soft_waitrdy**(struct *nand_chip* * *chip*, unsigned long *timeout_ms*)
Poll STATUS reg until RDY bit is set to 1

Parameters

struct nand_chip * chip NAND chip structure

unsigned long timeout_ms Timeout in ms

Description

Poll the STATUS register using ->:c:func:*exec_op()* until the RDY bit becomes 1. If that does not happen whitin the specified timeout, -ETIMEDOUT is returned.

This helper is intended to be used when the controller does not have access to the NAND R/B pin.

Be aware that calling this helper from an ->:c:func:*exec_op()* implementation means ->:c:func:*exec_op()* must be re-entrant.

Return 0 if the NAND chip is ready, a negative error otherwise.

int **nand_read_page_op**(struct *nand_chip* * *chip*, unsigned int *page*, unsigned int *offset_in_page*, void * *buf*, unsigned int *len*)
Do a READ PAGE operation

Parameters

struct nand_chip * chip The NAND chip

unsigned int page page to read

unsigned int offset_in_page offset within the page

void * buf buffer used to store the data

unsigned int len length of the buffer

Description

This function issues a READ PAGE operation. This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_change_read_column_op**(struct *nand_chip* * *chip*, unsigned int *offset_in_page*, void * *buf*, unsigned int *len*, bool *force_8bit*)
Do a CHANGE READ COLUMN operation

Parameters

struct nand_chip * chip The NAND chip

unsigned int offset_in_page offset within the page

void * buf buffer used to store the data

unsigned int len length of the buffer

bool force_8bit force 8-bit bus access

Description

This function issues a CHANGE READ COLUMN operation. This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_read_oob_op**(struct *nand_chip* * *chip*, unsigned int *page*, unsigned int *offset_in_oob*, void * *buf*, unsigned int *len*)
Do a READ OOB operation

Parameters

struct nand_chip * chip The NAND chip

unsigned int page page to read

unsigned int offset_in_oob offset within the OOB area

void * buf buffer used to store the data

unsigned int len length of the buffer

Description

This function issues a READ OOB operation. This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_prog_page_begin_op**(struct *nand_chip* * *chip*, unsigned int *page*, unsigned int *offset_in_page*, const void * *buf*, unsigned int *len*)
starts a PROG PAGE operation

Parameters

struct nand_chip * chip The NAND chip

unsigned int page page to write

unsigned int offset_in_page offset within the page

const void * buf buffer containing the data to write to the page

unsigned int len length of the buffer

Description

This function issues the first half of a PROG PAGE operation. This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_prog_page_end_op**(struct *nand_chip* * *chip*)
ends a PROG PAGE operation

Parameters

struct nand_chip * chip The NAND chip

Description

This function issues the second half of a PROG PAGE operation. This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_prog_page_op**(struct *nand_chip* * *chip*, unsigned int *page*, unsigned int *offset_in_page*, const void * *buf*, unsigned int *len*)
Do a full PROG PAGE operation

Parameters

struct nand_chip * chip The NAND chip

unsigned int page page to write

unsigned int offset_in_page offset within the page

const void * buf buffer containing the data to write to the page

unsigned int len length of the buffer

Description

This function issues a full PROG PAGE operation. This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_change_write_column_op**(struct *nand_chip* * *chip*, unsigned int *offset_in_page*, const void * *buf*, unsigned int *len*, bool *force_8bit*)
Do a CHANGE WRITE COLUMN operation

Parameters

struct nand_chip * chip The NAND chip

unsigned int offset_in_page offset within the page

const void * buf buffer containing the data to send to the NAND

unsigned int len length of the buffer

bool force_8bit force 8-bit bus access

Description

This function issues a CHANGE WRITE COLUMN operation. This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_readid_op**(struct *nand_chip* * *chip*, u8 *addr*, void * *buf*, unsigned int *len*)
Do a READID operation

Parameters

struct nand_chip * chip The NAND chip

u8 addr address cycle to pass after the READID command

void * buf buffer used to store the ID

unsigned int len length of the buffer

Description

This function sends a READID command and reads back the ID returned by the NAND. This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_status_op**(struct *nand_chip* * *chip*, u8 * *status*)
Do a STATUS operation

Parameters

struct nand_chip * chip The NAND chip

u8 * status out variable to store the NAND status

Description

This function sends a STATUS command and reads back the status returned by the NAND. This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_exit_status_op**(struct *nand_chip* * *chip*)
Exit a STATUS operation

Parameters

struct nand_chip * chip The NAND chip

Description

This function sends a READ0 command to cancel the effect of the STATUS command to avoid reading only the status until a new read command is sent.

This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_erase_op**(struct *nand_chip* * *chip*, unsigned int *eraseblock*)
    Do an erase operation

Parameters

struct nand_chip * chip The NAND chip

unsigned int eraseblock block to erase

Description

This function sends an ERASE command and waits for the NAND to be ready before returning. This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_reset_op**(struct *nand_chip* * *chip*)
    Do a reset operation

Parameters

struct nand_chip * chip The NAND chip

Description

This function sends a RESET command and waits for the NAND to be ready before returning. This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_read_data_op**(struct *nand_chip* * *chip*, void * *buf*, unsigned int *len*, bool *force_8bit*)
    Read data from the NAND

Parameters

struct nand_chip * chip The NAND chip

void * buf buffer used to store the data

unsigned int len length of the buffer

bool force_8bit force 8-bit bus access

Description

This function does a raw data read on the bus. Usually used after launching another NAND operation like *nand_read_page_op()*. This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_write_data_op**(struct *nand_chip* * *chip*, const void * *buf*, unsigned int *len*, bool *force_8bit*)
    Write data from the NAND

Parameters

struct nand_chip * chip The NAND chip

const void * buf buffer containing the data to send on the bus

unsigned int len length of the buffer

bool force_8bit force 8-bit bus access

Description

This function does a raw data write on the bus. Usually used after launching another NAND operation like `nand_write_page_begin_op()`. This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_op_parser_exec_op**(struct *nand_chip* * *chip*, const struct *nand_op_parser* * *parser*, const struct *nand_operation* * *op*, bool *check_only*)
exec_op parser

Parameters

struct nand_chip * chip the NAND chip

const struct nand_op_parser * parser patterns description provided by the controller driver

const struct nand_operation * op the NAND operation to address

bool check_only when true, the function only checks if **op** can be handled but does not execute the operation

Description

Helper function designed to ease integration of NAND controller drivers that only support a limited set of instruction sequences. The supported sequences are described in **parser**, and the framework takes care of splitting **op** into multiple sub-operations (if required) and pass them back to the ->:c:func:*exec()* callback of the matching pattern if **check_only** is set to false.

NAND controller drivers should call this function from their own ->:c:func:*exec_op()* implementation.

Returns 0 on success, a negative error code otherwise. A failure can be caused by an unsupported operation (none of the supported patterns is able to handle the requested operation), or an error returned by one of the matching pattern->:c:func:*exec()* hook.

int **nand_subop_get_addr_start_off**(const struct *nand_subop* * *subop*, unsigned int *instr_idx*)
Get the start offset in an address array

Parameters

const struct nand_subop * subop The entire sub-operation

unsigned int instr_idx Index of the instruction inside the sub-operation

Description

During driver development, one could be tempted to directly use the ->addr.addrs field of address instructions. This is wrong as address instructions might be split.

Given an address instruction, returns the offset of the first cycle to issue.

int **nand_subop_get_num_addr_cyc**(const struct *nand_subop* * *subop*, unsigned int *instr_idx*)
Get the remaining address cycles to assert

Parameters

const struct nand_subop * subop The entire sub-operation

unsigned int instr_idx Index of the instruction inside the sub-operation

Description

During driver development, one could be tempted to directly use the ->addr->naddrs field of a data instruction. This is wrong as instructions might be split.

Given an address instruction, returns the number of address cycle to issue.

int **nand_subop_get_data_start_off**(const struct *nand_subop* * *subop*, unsigned int *instr_idx*)
Get the start offset in a data array

Parameters

const struct nand_subop * subop The entire sub-operation

`unsigned int instr_idx` Index of the instruction inside the sub-operation

Description

During driver development, one could be tempted to directly use the ->data->buf.{in,out} field of data instructions. This is wrong as data instructions might be split.

Given a data instruction, returns the offset to start from.

int **`nand_subop_get_data_len`**(const struct *nand_subop* * *subop*, unsigned int *instr_idx*)
    Get the number of bytes to retrieve

Parameters

`const struct nand_subop * subop` The entire sub-operation

`unsigned int instr_idx` Index of the instruction inside the sub-operation

Description

During driver development, one could be tempted to directly use the ->data->len field of a data instruction. This is wrong as data instructions might be split.

Returns the length of the chunk of data to send/receive.

int **`nand_reset`**(struct *nand_chip* * *chip*, int *chipnr*)
    Reset and initialize a NAND device

Parameters

`struct nand_chip * chip` The NAND chip

`int chipnr` Internal die id

Description

Save the timings data structure, then apply SDR timings mode 0 (see nand_reset_data_interface for details), do the reset operation, and apply back the previous timings.

Returns 0 on success, a negative error code otherwise.

int **`nand_check_erased_ecc_chunk`**(void * *data*, int *datalen*, void * *ecc*, int *ecclen*, void * *extraoob*, int *extraooblen*, int *bitflips_threshold*)
    check if an ECC chunk contains (almost) only 0xff data

Parameters

`void * data` data buffer to test

`int datalen` data length

`void * ecc` ECC buffer

`int ecclen` ECC length

`void * extraoob` extra OOB buffer

`int extraooblen` extra OOB length

`int bitflips_threshold` maximum number of bitflips

Description

Check if a data buffer and its associated ECC and OOB data contains only 0xff pattern, which means the underlying region has been erased and is ready to be programmed. The bitflips_threshold specify the maximum number of bitflips before considering the region as not erased.

Note

1/ ECC algorithms are working on pre-defined block sizes which are usually different from the
    NAND page size. When fixing bitflips, ECC engines will report the number of errors per chunk, and the NAND core infrastructure expect you to return the maximum number of bitflips for the whole

page. This is why you should always use this function on a single chunk and not on the whole page. After checking each chunk you should update your max_bitflips value accordingly.

2/ When checking for bitflips in erased pages you should not only check the payload data but also their associated ECC data, because a user might have programmed almost all bits to 1 but a few. In this case, we shouldn't consider the chunk as erased, and checking ECC bytes prevent this case.

3/ The extraoob argument is optional, and should be used if some of your OOB data are protected by the ECC engine. It could also be used if you support subpages and want to attach some extra OOB data to an ECC chunk.

Returns a positive number of bitflips less than or equal to bitflips_threshold, or -ERROR_CODE for bitflips in excess of the threshold. In case of success, the passed buffers are filled with 0xff.

int **nand_read_page_raw**(struct mtd_info * *mtd*, struct nand_chip * *chip*, uint8_t * *buf*, int *oob_required*, int *page*)
[INTERN] read raw page data without ecc

Parameters

struct mtd_info * mtd mtd info structure

struct nand_chip * chip nand chip info structure

uint8_t * buf buffer to store read data

int oob_required caller requires OOB data read to chip->oob_poi

int page page number to read

Description

Not for syndrome calculating ECC controllers, which use a special oob layout.

int **nand_read_oob_std**(struct mtd_info * *mtd*, struct nand_chip * *chip*, int *page*)
[REPLACEABLE] the most common OOB data read function

Parameters

struct mtd_info * mtd mtd info structure

struct nand_chip * chip nand chip info structure

int page page number to read

int **nand_read_oob_syndrome**(struct mtd_info * *mtd*, struct nand_chip * *chip*, int *page*)
[REPLACEABLE] OOB data read function for HW ECC with syndromes

Parameters

struct mtd_info * mtd mtd info structure

struct nand_chip * chip nand chip info structure

int page page number to read

int **nand_write_oob_std**(struct mtd_info * *mtd*, struct nand_chip * *chip*, int *page*)
[REPLACEABLE] the most common OOB data write function

Parameters

struct mtd_info * mtd mtd info structure

struct nand_chip * chip nand chip info structure

int page page number to write

int **nand_write_oob_syndrome**(struct mtd_info * *mtd*, struct nand_chip * *chip*, int *page*)
[REPLACEABLE] OOB data write function for HW ECC with syndrome - only for large page flash

Parameters

struct mtd_info * mtd mtd info structure

struct nand_chip * chip nand chip info structure

int page page number to write

int **nand_write_page_raw**(struct mtd_info * *mtd*, struct nand_chip * *chip*, const uint8_t * *buf*, int *oob_required*, int *page*)
[INTERN] raw page write function

Parameters

struct mtd_info * mtd mtd info structure

struct nand_chip * chip nand chip info structure

const uint8_t * buf data buffer

int oob_required must write chip->oob_poi to OOB

int page page number to write

Description

Not for syndrome calculating ECC controllers, which use a special oob layout.

int **nand_onfi_get_set_features_notsupp**(struct mtd_info * *mtd*, struct nand_chip * *chip*, int *addr*, u8 * *subfeature_param*)
set/get features stub returning -ENOTSUPP

Parameters

struct mtd_info * mtd MTD device structure

struct nand_chip * chip nand chip info structure

int addr feature address.

u8 * subfeature_param the subfeature parameters, a four bytes array.

Description

Should be used by NAND controller drivers that do not support the SET/GET FEATURES operations.

int **nand_scan_ident**(struct mtd_info * *mtd*, int *maxchips*, struct nand_flash_dev * *table*)
[NAND Interface] Scan for the NAND device

Parameters

struct mtd_info * mtd MTD device structure

int maxchips number of chips to scan for

struct nand_flash_dev * table alternative NAND ID table

Description

This is the first phase of the normal *nand_scan()* function. It reads the flash ID and sets up MTD fields accordingly.

int **nand_check_ecc_caps**(struct nand_chip * *chip*, const struct nand_ecc_caps * *caps*, int *oobavail*)
check the sanity of preset ECC settings

Parameters

struct nand_chip * chip nand chip info structure

const struct nand_ecc_caps * caps ECC caps info structure

int oobavail OOB size that the ECC engine can use

Description

When ECC step size and strength are already set, check if they are supported by the controller and the calculated ECC bytes fit within the chip's OOB. On success, the calculated ECC bytes is set.

int **nand_match_ecc_req**(struct *nand_chip* * *chip*, const struct *nand_ecc_caps* * *caps*, int *oobavail*)
meet the chip's requirement with least ECC bytes

Parameters

struct nand_chip * chip nand chip info structure

const struct nand_ecc_caps * caps ECC engine caps info structure

int oobavail OOB size that the ECC engine can use

Description

If a chip's ECC requirement is provided, try to meet it with the least number of ECC bytes (i.e. with the largest number of OOB-free bytes). On success, the chosen ECC settings are set.

int **nand_maximize_ecc**(struct *nand_chip* * *chip*, const struct *nand_ecc_caps* * *caps*, int *oobavail*)
choose the max ECC strength available

Parameters

struct nand_chip * chip nand chip info structure

const struct nand_ecc_caps * caps ECC engine caps info structure

int oobavail OOB size that the ECC engine can use

Description

Choose the max ECC strength that is supported on the controller, and can fit within the chip's OOB. On success, the chosen ECC settings are set.

int **nand_scan_tail**(struct mtd_info * *mtd*)
[NAND Interface] Scan for the NAND device

Parameters

struct mtd_info * mtd MTD device structure

Description

This is the second phase of the normal *nand_scan()* function. It fills out all the uninitialized function pointers with the defaults and scans for a bad block table if appropriate.

int **nand_scan**(struct mtd_info * *mtd*, int *maxchips*)
[NAND Interface] Scan for the NAND device

Parameters

struct mtd_info * mtd MTD device structure

int maxchips number of chips to scan for

Description

This fills out all the uninitialized function pointers with the defaults. The flash ID is read and the mtd/chip structures are filled with the appropriate values.

void **nand_cleanup**(struct *nand_chip* * *chip*)
[NAND Interface] Free resources held by the NAND device

Parameters

struct nand_chip * chip NAND chip object

void **nand_release**(struct mtd_info * *mtd*)
[NAND Interface] Unregister the MTD device and free resources held by the NAND device

Parameters

struct mtd_info * mtd MTD device structure

void **__nand_calculate_ecc**(const unsigned char * *buf*, unsigned int *eccsize*, unsigned char * *code*)
[NAND Interface] Calculate 3-byte ECC for 256/512-byte block

Parameters

`const unsigned char * buf` input buffer with raw data

`unsigned int eccsize` data bytes per ECC step (256 or 512)

`unsigned char * code` output buffer with ECC

int **`nand_calculate_ecc`**(struct mtd_info * *mtd*, const unsigned char * *buf*, unsigned char * *code*)
[NAND Interface] Calculate 3-byte ECC for 256/512-byte block

Parameters

`struct mtd_info * mtd` MTD block structure

`const unsigned char * buf` input buffer with raw data

`unsigned char * code` output buffer with ECC

int **`__nand_correct_data`**(unsigned char * *buf*, unsigned char * *read_ecc*, unsigned char * *calc_ecc*, unsigned int *eccsize*)
[NAND Interface] Detect and correct bit error(s)

Parameters

`unsigned char * buf` raw data read from the chip

`unsigned char * read_ecc` ECC from the chip

`unsigned char * calc_ecc` the ECC calculated from raw data

`unsigned int eccsize` data bytes per ECC step (256 or 512)

Description

Detect and correct a 1 bit error for eccsize byte block

int **`nand_correct_data`**(struct mtd_info * *mtd*, unsigned char * *buf*, unsigned char * *read_ecc*, unsigned char * *calc_ecc*)
[NAND Interface] Detect and correct bit error(s)

Parameters

`struct mtd_info * mtd` MTD block structure

`unsigned char * buf` raw data read from the chip

`unsigned char * read_ecc` ECC from the chip

`unsigned char * calc_ecc` the ECC calculated from raw data

Description

Detect and correct a 1 bit error for 256/512 byte block

Internal Functions Provided

This chapter contains the autogenerated documentation of the NAND driver internal functions. Each function has a short description which is marked with an [XXX] identifier. See the chapter "Documentation hints" for an explanation. The functions marked with [DEFAULT] might be relevant for a board driver developer.

void **`nand_release_device`**(struct mtd_info * *mtd*)
[GENERIC] release chip

Parameters

`struct mtd_info * mtd` MTD device structure

Description

Release chip lock and wake up anyone waiting on the device.

uint8_t **nand_read_byte**(struct mtd_info * *mtd*)
[DEFAULT] read one byte from the chip

Parameters

struct mtd_info * mtd MTD device structure

Description

Default read function for 8bit buswidth

uint8_t **nand_read_byte16**(struct mtd_info * *mtd*)
[DEFAULT] read one byte endianness aware from the chip

Parameters

struct mtd_info * mtd MTD device structure

Description

Default read function for 16bit buswidth with endianness conversion.

u16 **nand_read_word**(struct mtd_info * *mtd*)
[DEFAULT] read one word from the chip

Parameters

struct mtd_info * mtd MTD device structure

Description

Default read function for 16bit buswidth without endianness conversion.

void **nand_select_chip**(struct mtd_info * *mtd*, int *chipnr*)
[DEFAULT] control CE line

Parameters

struct mtd_info * mtd MTD device structure

int chipnr chipnumber to select, -1 for deselect

Description

Default select function for 1 chip devices.

void **nand_write_byte**(struct mtd_info * *mtd*, uint8_t *byte*)
[DEFAULT] write single byte to chip

Parameters

struct mtd_info * mtd MTD device structure

uint8_t byte value to write

Description

Default function to write a byte to I/O[7:0]

void **nand_write_byte16**(struct mtd_info * *mtd*, uint8_t *byte*)
[DEFAULT] write single byte to a chip with width 16

Parameters

struct mtd_info * mtd MTD device structure

uint8_t byte value to write

Description

Default function to write a byte to I/O[7:0] on a 16-bit wide chip.

void **nand_write_buf**(struct mtd_info * *mtd*, const uint8_t * *buf*, int *len*)
[DEFAULT] write buffer to chip

Parameters

struct mtd_info * mtd MTD device structure

const uint8_t * buf data buffer

int len number of bytes to write

Description

Default write function for 8bit buswidth.

void **nand_read_buf**(struct mtd_info * *mtd*, uint8_t * *buf*, int *len*)
[DEFAULT] read chip data into buffer

Parameters

struct mtd_info * mtd MTD device structure

uint8_t * buf buffer to store date

int len number of bytes to read

Description

Default read function for 8bit buswidth.

void **nand_write_buf16**(struct mtd_info * *mtd*, const uint8_t * *buf*, int *len*)
[DEFAULT] write buffer to chip

Parameters

struct mtd_info * mtd MTD device structure

const uint8_t * buf data buffer

int len number of bytes to write

Description

Default write function for 16bit buswidth.

void **nand_read_buf16**(struct mtd_info * *mtd*, uint8_t * *buf*, int *len*)
[DEFAULT] read chip data into buffer

Parameters

struct mtd_info * mtd MTD device structure

uint8_t * buf buffer to store date

int len number of bytes to read

Description

Default read function for 16bit buswidth.

int **nand_block_bad**(struct mtd_info * *mtd*, loff_t *ofs*)
[DEFAULT] Read bad block marker from the chip

Parameters

struct mtd_info * mtd MTD device structure

loff_t ofs offset from device start

Description

Check, if the block is bad.

int **nand_default_block_markbad**(struct mtd_info * *mtd*, loff_t *ofs*)
[DEFAULT] mark a block bad via bad block marker

Parameters

struct mtd_info * mtd MTD device structure

loff_t ofs offset from device start

Description

This is the default implementation, which can be overridden by a hardware specific driver. It provides the details for writing a bad block marker to a block.

int **nand_block_markbad_lowlevel**(struct mtd_info * *mtd*, loff_t *ofs*)
mark a block bad

Parameters

struct mtd_info * mtd MTD device structure

loff_t ofs offset from device start

Description

This function performs the generic NAND bad block marking steps (i.e., bad block table(s) and/or marker(s)). We only allow the hardware driver to specify how to write bad block markers to OOB (chip->block_markbad).

We try operations in the following order:

1. erase the affected block, to allow OOB marker to be written cleanly
2. write bad block marker to OOB area of affected block (unless flag NAND_BBT_NO_OOB_BBM is present)
3. update the BBT

Note that we retain the first error encountered in (2) or (3), finish the procedures, and dump the error in the end.

int **nand_check_wp**(struct mtd_info * *mtd*)
[GENERIC] check if the chip is write protected

Parameters

struct mtd_info * mtd MTD device structure

Description

Check, if the device is write protected. The function expects, that the device is already selected.

int **nand_block_isreserved**(struct mtd_info * *mtd*, loff_t *ofs*)
[GENERIC] Check if a block is marked reserved.

Parameters

struct mtd_info * mtd MTD device structure

loff_t ofs offset from device start

Description

Check if the block is marked as reserved.

int **nand_block_checkbad**(struct mtd_info * *mtd*, loff_t *ofs*, int *allowbbt*)
[GENERIC] Check if a block is marked bad

Parameters

struct mtd_info * mtd MTD device structure

loff_t ofs offset from device start

int allowbbt 1, if its allowed to access the bbt area

Description

Check, if the block is bad. Either by reading the bad block table or calling of the scan function.

void **panic_nand_wait_ready**(struct mtd_info * *mtd*, unsigned long *timeo*)
[GENERIC] Wait for the ready pin after commands.

Parameters

struct mtd_info * mtd MTD device structure

unsigned long timeo Timeout

Description

Helper function for nand_wait_ready used when needing to wait in interrupt context.

void **nand_wait_status_ready**(struct mtd_info * *mtd*, unsigned long *timeo*)
[GENERIC] Wait for the ready status after commands.

Parameters

struct mtd_info * mtd MTD device structure

unsigned long timeo Timeout in ms

Description

Wait for status ready (i.e. command done) or timeout.

void **nand_command**(struct mtd_info * *mtd*, unsigned int *command*, int *column*, int *page_addr*)
[DEFAULT] Send command to NAND device

Parameters

struct mtd_info * mtd MTD device structure

unsigned int command the command to be sent

int column the column address for this command, -1 if none

int page_addr the page address for this command, -1 if none

Description

Send command to NAND device. This function is used for small page devices (512 Bytes per page).

void **nand_command_lp**(struct mtd_info * *mtd*, unsigned int *command*, int *column*, int *page_addr*)
[DEFAULT] Send command to NAND large page device

Parameters

struct mtd_info * mtd MTD device structure

unsigned int command the command to be sent

int column the column address for this command, -1 if none

int page_addr the page address for this command, -1 if none

Description

Send command to NAND device. This is the version for the new large page devices. We don't have the separate regions as we have in the small page devices. We must emulate NAND_CMD_READOOB to keep the code compatible.

void **panic_nand_get_device**(struct *nand_chip* * *chip*, struct mtd_info * *mtd*, int *new_state*)
[GENERIC] Get chip for selected access

Parameters

struct nand_chip * chip the nand chip descriptor

struct mtd_info * mtd MTD device structure

int new_state the state which is requested

Description

Used when in panic, no locks are taken.

int **nand_get_device**(struct mtd_info * *mtd*, int *new_state*)
[GENERIC] Get chip for selected access

Parameters

struct mtd_info * mtd MTD device structure

int new_state the state which is requested

Description

Get the device and lock it for exclusive access

void **panic_nand_wait**(struct mtd_info * *mtd*, struct *nand_chip* * *chip*, unsigned long *timeo*)
[GENERIC] wait until the command is done

Parameters

struct mtd_info * mtd MTD device structure

struct nand_chip * chip NAND chip structure

unsigned long timeo timeout

Description

Wait for command done. This is a helper function for nand_wait used when we are in interrupt context. May happen when in panic and trying to write an oops through mtdoops.

int **nand_wait**(struct mtd_info * *mtd*, struct *nand_chip* * *chip*)
[DEFAULT] wait until the command is done

Parameters

struct mtd_info * mtd MTD device structure

struct nand_chip * chip NAND chip structure

Description

Wait for command done. This applies to erase and program only.

int **nand_reset_data_interface**(struct *nand_chip* * *chip*, int *chipnr*)
Reset data interface and timings

Parameters

struct nand_chip * chip The NAND chip

int chipnr Internal die id

Description

Reset the Data interface and timings to ONFI mode 0.

Returns 0 for success or negative error code otherwise.

int **nand_setup_data_interface**(struct *nand_chip* * *chip*, int *chipnr*)
Setup the best data interface and timings

Parameters

struct nand_chip * chip The NAND chip

int chipnr Internal die id

Description

Find and configure the best data interface and NAND timings supported by the chip and the driver. First tries to retrieve supported timing modes from ONFI information, and if the NAND chip does not support ONFI, relies on the ->onfi_timing_mode_default specified in the nand_ids table.

Returns 0 for success or negative error code otherwise.

int **nand_init_data_interface**(struct *nand_chip* * *chip*)
find the best data interface and timings

Parameters

struct nand_chip * chip The NAND chip

Description

Find the best data interface and NAND timings supported by the chip and the driver. First tries to retrieve supported timing modes from ONFI information, and if the NAND chip does not support ONFI, relies on the ->onfi_timing_mode_default specified in the nand_ids table. After this function nand_chip->data_interface is initialized with the best timing mode available.

Returns 0 for success or negative error code otherwise.

int **nand_fill_column_cycles**(struct *nand_chip* * *chip*, u8 * *addrs*, unsigned int *offset_in_page*)
fill the column cycles of an address

Parameters

struct nand_chip * chip The NAND chip

u8 * addrs Array of address cycles to fill

unsigned int offset_in_page The offset in the page

Description

Fills the first or the first two bytes of the **addrs** field depending on the NAND bus width and the page size.

Returns the number of cycles needed to encode the column, or a negative error code in case one of the arguments is invalid.

int **nand_read_param_page_op**(struct *nand_chip* * *chip*, u8 *page*, void * *buf*, unsigned int *len*)
Do a READ PARAMETER PAGE operation

Parameters

struct nand_chip * chip The NAND chip

u8 page parameter page to read

void * buf buffer used to store the data

unsigned int len length of the buffer

Description

This function issues a READ PARAMETER PAGE operation. This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_set_features_op**(struct *nand_chip* * *chip*, u8 *feature*, const void * *data*)
Do a SET FEATURES operation

Parameters

struct nand_chip * chip The NAND chip

u8 feature feature id

const void * data 4 bytes of data

Description

This function sends a SET FEATURES command and waits for the NAND to be ready before returning. This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_get_features_op**(struct *nand_chip* * *chip*, u8 *feature*, void * *data*)
Do a GET FEATURES operation

Parameters

struct nand_chip * chip The NAND chip

u8 feature feature id

void * data 4 bytes of data

Description

This function sends a GET FEATURES command and waits for the NAND to be ready before returning. This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

struct **nand_op_parser_ctx**
Context used by the parser

Definition

```
struct nand_op_parser_ctx {
  const struct nand_op_instr *instrs;
  unsigned int ninstrs;
  struct nand_subop subop;
};
```

Members

instrs array of all the instructions that must be addressed

ninstrs length of the **instrs** array

subop Sub-operation to be passed to the NAND controller

Description

This structure is used by the core to split NAND operations into sub-operations that can be handled by the NAND controller.

bool **nand_op_parser_must_split_instr**(const struct *nand_op_parser_pattern_elem* * *pat*, const struct *nand_op_instr* * *instr*, unsigned int * *start_offset*)
Checks if an instruction must be split

Parameters

const struct nand_op_parser_pattern_elem * pat the parser pattern element that matches **instr**

const struct nand_op_instr * instr pointer to the instruction to check

unsigned int * start_offset this is an in/out parameter. If **instr** has already been split, then **start_offset** is the offset from which to start (either an address cycle or an offset in the data buffer). Conversely, if the function returns true (ie. instr must be split), this parameter is updated to point to the first data/address cycle that has not been taken care of.

Description

Some NAND controllers are limited and cannot send X address cycles with a unique operation, or cannot read/write more than Y bytes at the same time. In this case, split the instruction that does not fit in a single controller-operation into two or more chunks.

Returns true if the instruction must be split, false otherwise. The **start_offset** parameter is also updated to the offset at which the next bundle of instruction must start (if an address or a data instruction).

bool **nand_op_parser_match_pat**(const struct *nand_op_parser_pattern* * *pat*, struct *nand_op_parser_ctx* * *ctx*)
Checks if a pattern matches the instructions remaining in the parser context

Parameters

const struct nand_op_parser_pattern * pat the pattern to test

struct nand_op_parser_ctx * ctx the parser context structure to match with the pattern **pat**

Description

Check if **pat** matches the set or a sub-set of instructions remaining in **ctx**. Returns true if this is the case, false ortherwise. When true is returned, **ctx**->subop is updated with the set of instructions to be passed to the controller driver.

int **nand_check_erased_buf**(void * *buf*, int *len*, int *bitflips_threshold*)
check if a buffer contains (almost) only 0xff data

Parameters

void * buf buffer to test

int len buffer length

int bitflips_threshold maximum number of bitflips

Description

Check if a buffer contains only 0xff, which means the underlying region has been erased and is ready to be programmed. The bitflips_threshold specify the maximum number of bitflips before considering the region is not erased.

Note

The logic of this function has been extracted from the memweight implementation, except that nand_check_erased_buf function exit before testing the whole buffer if the number of bitflips exceed the bitflips_threshold value.

Returns a positive number of bitflips less than or equal to bitflips_threshold, or -ERROR_CODE for bitflips in excess of the threshold.

int **nand_read_page_raw_syndrome**(struct mtd_info * *mtd*, struct *nand_chip* * *chip*, uint8_t * *buf*, int *oob_required*, int *page*)
[INTERN] read raw page data without ecc

Parameters

struct mtd_info * mtd mtd info structure

struct nand_chip * chip nand chip info structure

uint8_t * buf buffer to store read data

int oob_required caller requires OOB data read to chip->oob_poi

int page page number to read

Description

We need a special oob layout and handling even when OOB isn't used.

int **nand_read_page_swecc**(struct mtd_info * *mtd*, struct *nand_chip* * *chip*, uint8_t * *buf*, int *oob_required*, int *page*)
[REPLACEABLE] software ECC based page read function

Parameters

struct mtd_info * mtd mtd info structure

struct nand_chip * chip nand chip info structure

uint8_t * buf buffer to store read data

int oob_required caller requires OOB data read to chip->oob_poi

int page page number to read

int **nand_read_subpage**(struct mtd_info * *mtd*, struct nand_chip * *chip*, uint32_t *data_offs*, uint32_t *readlen*, uint8_t * *bufpoi*, int *page*)
[REPLACEABLE] ECC based sub-page read function

Parameters

struct mtd_info * mtd mtd info structure

struct nand_chip * chip nand chip info structure

uint32_t data_offs offset of requested data within the page

uint32_t readlen data length

uint8_t * bufpoi buffer to store read data

int page page number to read

int **nand_read_page_hwecc**(struct mtd_info * *mtd*, struct nand_chip * *chip*, uint8_t * *buf*, int *oob_required*, int *page*)
[REPLACEABLE] hardware ECC based page read function

Parameters

struct mtd_info * mtd mtd info structure

struct nand_chip * chip nand chip info structure

uint8_t * buf buffer to store read data

int oob_required caller requires OOB data read to chip->oob_poi

int page page number to read

Description

Not for syndrome calculating ECC controllers which need a special oob layout.

int **nand_read_page_hwecc_oob_first**(struct mtd_info * *mtd*, struct nand_chip * *chip*, uint8_t * *buf*, int *oob_required*, int *page*)
[REPLACEABLE] hw ecc, read oob first

Parameters

struct mtd_info * mtd mtd info structure

struct nand_chip * chip nand chip info structure

uint8_t * buf buffer to store read data

int oob_required caller requires OOB data read to chip->oob_poi

int page page number to read

Description

Hardware ECC for large page chips, require OOB to be read first. For this ECC mode, the write_page method is re-used from ECC_HW. These methods read/write ECC from the OOB area, unlike the ECC_HW_SYNDROME support with multiple ECC steps, follows the "infix ECC" scheme and reads/writes ECC from the data area, by overwriting the NAND manufacturer bad block markings.

int **nand_read_page_syndrome**(struct mtd_info * *mtd*, struct nand_chip * *chip*, uint8_t * *buf*, int *oob_required*, int *page*)
[REPLACEABLE] hardware ECC syndrome based page read

Parameters

struct mtd_info * mtd mtd info structure

struct nand_chip * chip nand chip info structure

uint8_t * buf buffer to store read data

int oob_required caller requires OOB data read to chip->oob_poi

int page page number to read

Description

The hw generator calculates the error syndrome automatically. Therefore we need a special oob layout and handling.

uint8_t * **nand_transfer_oob**(struct mtd_info * *mtd*, uint8_t * *oob*, struct mtd_oob_ops * *ops*, size_t *len*)
[INTERN] Transfer oob to client buffer

Parameters

struct mtd_info * mtd mtd info structure

uint8_t * oob oob destination address

struct mtd_oob_ops * ops oob ops structure

size_t len size of oob to transfer

int **nand_setup_read_retry**(struct mtd_info * *mtd*, int *retry_mode*)
[INTERN] Set the READ RETRY mode

Parameters

struct mtd_info * mtd MTD device structure

int retry_mode the retry mode to use

Description

Some vendors supply a special command to shift the Vt threshold, to be used when there are too many bitflips in a page (i.e., ECC error). After setting a new threshold, the host should retry reading the page.

int **nand_do_read_ops**(struct mtd_info * *mtd*, loff_t *from*, struct mtd_oob_ops * *ops*)
[INTERN] Read data with ECC

Parameters

struct mtd_info * mtd MTD device structure

loff_t from offset to read from

struct mtd_oob_ops * ops oob ops structure

Description

Internal function. Called with chip held.

int **nand_do_read_oob**(struct mtd_info * *mtd*, loff_t *from*, struct mtd_oob_ops * *ops*)
[INTERN] NAND read out-of-band

Parameters

struct mtd_info * mtd MTD device structure

loff_t from offset to read from

struct mtd_oob_ops * ops oob operations description structure

Description

NAND read out-of-band data from the spare area.

int **nand_read_oob**(struct mtd_info * *mtd*, loff_t *from*, struct mtd_oob_ops * *ops*)
[MTD Interface] NAND read data and/or out-of-band

Parameters

struct mtd_info * mtd MTD device structure

loff_t from offset to read from

struct mtd_oob_ops * ops oob operation description structure

Description

NAND read data and/or out-of-band data.

int **nand_write_page_raw_syndrome**(struct mtd_info * *mtd*, struct *nand_chip* * *chip*, const uint8_t * *buf*, int *oob_required*, int *page*)
    [INTERN] raw page write function

Parameters

struct mtd_info * mtd mtd info structure

struct nand_chip * chip nand chip info structure

const uint8_t * buf data buffer

int oob_required must write chip->oob_poi to OOB

int page page number to write

Description

We need a special oob layout and handling even when ECC isn't checked.

int **nand_write_page_swecc**(struct mtd_info * *mtd*, struct *nand_chip* * *chip*, const uint8_t * *buf*, int *oob_required*, int *page*)
    [REPLACEABLE] software ECC based page write function

Parameters

struct mtd_info * mtd mtd info structure

struct nand_chip * chip nand chip info structure

const uint8_t * buf data buffer

int oob_required must write chip->oob_poi to OOB

int page page number to write

int **nand_write_page_hwecc**(struct mtd_info * *mtd*, struct *nand_chip* * *chip*, const uint8_t * *buf*, int *oob_required*, int *page*)
    [REPLACEABLE] hardware ECC based page write function

Parameters

struct mtd_info * mtd mtd info structure

struct nand_chip * chip nand chip info structure

const uint8_t * buf data buffer

int oob_required must write chip->oob_poi to OOB

int page page number to write

int **nand_write_subpage_hwecc**(struct mtd_info * *mtd*, struct *nand_chip* * *chip*, uint32_t *offset*, uint32_t *data_len*, const uint8_t * *buf*, int *oob_required*, int *page*)
    [REPLACEABLE] hardware ECC based subpage write

Parameters

struct mtd_info * mtd mtd info structure

struct nand_chip * chip nand chip info structure

uint32_t offset column address of subpage within the page

uint32_t data_len data length

const uint8_t * buf data buffer

int oob_required must write chip->oob_poi to OOB

int page page number to write

int **nand_write_page_syndrome**(struct mtd_info * *mtd*, struct nand_chip * *chip*, const uint8_t * *buf*, int *oob_required*, int *page*)
[REPLACEABLE] hardware ECC syndrome based page write

Parameters

struct mtd_info * mtd mtd info structure

struct nand_chip * chip nand chip info structure

const uint8_t * buf data buffer

int oob_required must write chip->oob_poi to OOB

int page page number to write

Description

The hw generator calculates the error syndrome automatically. Therefore we need a special oob layout and handling.

int **nand_write_page**(struct mtd_info * *mtd*, struct nand_chip * *chip*, uint32_t *offset*, int *data_len*, const uint8_t * *buf*, int *oob_required*, int *page*, int *raw*)
write one page

Parameters

struct mtd_info * mtd MTD device structure

struct nand_chip * chip NAND chip descriptor

uint32_t offset address offset within the page

int data_len length of actual data to be written

const uint8_t * buf the data to write

int oob_required must write chip->oob_poi to OOB

int page page number to write

int raw use _raw version of write_page

uint8_t * **nand_fill_oob**(struct mtd_info * *mtd*, uint8_t * *oob*, size_t *len*, struct mtd_oob_ops * *ops*)
[INTERN] Transfer client buffer to oob

Parameters

struct mtd_info * mtd MTD device structure

uint8_t * oob oob data buffer

size_t len oob data write length

struct mtd_oob_ops * ops oob ops structure

int **nand_do_write_ops**(struct mtd_info * *mtd*, loff_t *to*, struct mtd_oob_ops * *ops*)
[INTERN] NAND write with ECC

Parameters

struct mtd_info * mtd MTD device structure

loff_t to offset to write to

struct mtd_oob_ops * ops oob operations description structure

Description

NAND write with ECC.

int **panic_nand_write**(struct mtd_info * *mtd*, loff_t *to*, size_t *len*, size_t * *retlen*, const uint8_t * *buf*)
[MTD Interface] NAND write with ECC

Parameters

struct mtd_info * mtd MTD device structure

loff_t to offset to write to

size_t len number of bytes to write

size_t * retlen pointer to variable to store the number of written bytes

const uint8_t * buf the data to write

Description

NAND write with ECC. Used when performing writes in interrupt context, this may for example be called by mtdoops when writing an oops while in panic.

int **nand_do_write_oob**(struct mtd_info * *mtd*, loff_t *to*, struct mtd_oob_ops * *ops*)
[MTD Interface] NAND write out-of-band

Parameters

struct mtd_info * mtd MTD device structure

loff_t to offset to write to

struct mtd_oob_ops * ops oob operation description structure

Description

NAND write out-of-band.

int **nand_write_oob**(struct mtd_info * *mtd*, loff_t *to*, struct mtd_oob_ops * *ops*)
[MTD Interface] NAND write data and/or out-of-band

Parameters

struct mtd_info * mtd MTD device structure

loff_t to offset to write to

struct mtd_oob_ops * ops oob operation description structure

int **single_erase**(struct mtd_info * *mtd*, int *page*)
[GENERIC] NAND standard block erase command function

Parameters

struct mtd_info * mtd MTD device structure

int page the page address of the block which will be erased

Description

Standard erase command for NAND chips. Returns NAND status.

int **nand_erase**(struct mtd_info * *mtd*, struct erase_info * *instr*)
[MTD Interface] erase block(s)

Parameters

struct mtd_info * mtd MTD device structure

struct erase_info * instr erase instruction

Description

Erase one ore more blocks.

int **nand_erase_nand**(struct mtd_info * *mtd*, struct erase_info * *instr*, int *allowbbt*)
[INTERN] erase block(s)

Parameters

struct mtd_info * mtd MTD device structure

struct erase_info * instr erase instruction

int allowbbt allow erasing the bbt area

Description

Erase one ore more blocks.

void **nand_sync**(struct mtd_info * *mtd*)
[MTD Interface] sync

Parameters

struct mtd_info * mtd MTD device structure

Description

Sync is actually a wait for chip ready function.

int **nand_block_isbad**(struct mtd_info * *mtd*, loff_t *offs*)
[MTD Interface] Check if block at offset is bad

Parameters

struct mtd_info * mtd MTD device structure

loff_t offs offset relative to mtd start

int **nand_block_markbad**(struct mtd_info * *mtd*, loff_t *ofs*)
[MTD Interface] Mark block at the given offset as bad

Parameters

struct mtd_info * mtd MTD device structure

loff_t ofs offset relative to mtd start

int **nand_max_bad_blocks**(struct mtd_info * *mtd*, loff_t *ofs*, size_t *len*)
[MTD Interface] Max number of bad blocks for an mtd

Parameters

struct mtd_info * mtd MTD device structure

loff_t ofs offset relative to mtd start

size_t len length of mtd

int **nand_onfi_set_features**(struct mtd_info * *mtd*, struct *nand_chip* * *chip*, int *addr*, uint8_t * *sub-feature_param*)
[REPLACEABLE] set features for ONFI nand

Parameters

struct mtd_info * mtd MTD device structure

struct nand_chip * chip nand chip info structure

int addr feature address.

uint8_t * subfeature_param the subfeature parameters, a four bytes array.

int **nand_onfi_get_features**(struct mtd_info * *mtd*, struct *nand_chip* * *chip*, int *addr*, uint8_t * *sub-feature_param*)
[REPLACEABLE] get features for ONFI nand

Parameters

struct mtd_info * mtd MTD device structure

struct nand_chip * chip nand chip info structure

int addr feature address.

uint8_t * subfeature_param the subfeature parameters, a four bytes array.

int **nand_suspend**(struct mtd_info * *mtd*)
    [MTD Interface] Suspend the NAND flash

Parameters

struct mtd_info * mtd MTD device structure

void **nand_resume**(struct mtd_info * *mtd*)
    [MTD Interface] Resume the NAND flash

Parameters

struct mtd_info * mtd MTD device structure

void **nand_shutdown**(struct mtd_info * *mtd*)
    [MTD Interface] Finish the current NAND operation and prevent further operations

Parameters

struct mtd_info * mtd MTD device structure

int **check_pattern**(uint8_t * *buf*, int *len*, int *paglen*, struct nand_bbt_descr * *td*)
    [GENERIC] check if a pattern is in the buffer

Parameters

uint8_t * buf the buffer to search

int len the length of buffer to search

int paglen the pagelength

struct nand_bbt_descr * td search pattern descriptor

Description

Check for a pattern at the given place. Used to search bad block tables and good / bad block identifiers.

int **check_short_pattern**(uint8_t * *buf*, struct nand_bbt_descr * *td*)
    [GENERIC] check if a pattern is in the buffer

Parameters

uint8_t * buf the buffer to search

struct nand_bbt_descr * td search pattern descriptor

Description

Check for a pattern at the given place. Used to search bad block tables and good / bad block identifiers. Same as check_pattern, but no optional empty check.

u32 **add_marker_len**(struct nand_bbt_descr * *td*)
    compute the length of the marker in data area

Parameters

struct nand_bbt_descr * td BBT descriptor used for computation

Description

The length will be 0 if the marker is located in OOB area.

int **read_bbt**(struct mtd_info * *mtd*, uint8_t * *buf*, int *page*, int *num*, struct nand_bbt_descr * *td*, int *offs*)
    [GENERIC] Read the bad block table starting from page

Parameters

struct mtd_info * mtd MTD device structure

uint8_t * buf temporary buffer

int page the starting page

int num the number of bbt descriptors to read

struct nand_bbt_descr * td the bbt describtion table

int offs block number offset in the table

Description

Read the bad block table starting from page.

int **read_abs_bbt**(struct mtd_info * *mtd*, uint8_t * *buf*, struct nand_bbt_descr * *td*, int *chip*)
[GENERIC] Read the bad block table starting at a given page

Parameters

struct mtd_info * mtd MTD device structure

uint8_t * buf temporary buffer

struct nand_bbt_descr * td descriptor for the bad block table

int chip read the table for a specific chip, -1 read all chips; applies only if NAND_BBT_PERCHIP option is set

Description

Read the bad block table for all chips starting at a given page. We assume that the bbt bits are in consecutive order.

int **scan_read_oob**(struct mtd_info * *mtd*, uint8_t * *buf*, loff_t *offs*, size_t *len*)
[GENERIC] Scan data+OOB region to buffer

Parameters

struct mtd_info * mtd MTD device structure

uint8_t * buf temporary buffer

loff_t offs offset at which to scan

size_t len length of data region to read

Description

Scan read data from data+OOB. May traverse multiple pages, interleaving page,OOB,page,OOB,... in buf. Completes transfer and returns the "strongest" ECC condition (error or bitflip). May quit on the first (non-ECC) error.

void **read_abs_bbts**(struct mtd_info * *mtd*, uint8_t * *buf*, struct nand_bbt_descr * *td*, struct nand_bbt_descr * *md*)
[GENERIC] Read the bad block table(s) for all chips starting at a given page

Parameters

struct mtd_info * mtd MTD device structure

uint8_t * buf temporary buffer

struct nand_bbt_descr * td descriptor for the bad block table

struct nand_bbt_descr * md descriptor for the bad block table mirror

Description

Read the bad block table(s) for all chips starting at a given page. We assume that the bbt bits are in consecutive order.

int **create_bbt**(struct mtd_info * *mtd*, uint8_t * *buf*, struct nand_bbt_descr * *bd*, int *chip*)
[GENERIC] Create a bad block table by scanning the device

Parameters

struct mtd_info * mtd MTD device structure

uint8_t * buf temporary buffer

struct nand_bbt_descr * bd descriptor for the good/bad block search pattern

int chip create the table for a specific chip, -1 read all chips; applies only if NAND_BBT_PERCHIP option is set

Description

Create a bad block table by scanning the device for the given good/bad block identify pattern.

int **search_bbt**(struct mtd_info * *mtd*, uint8_t * *buf*, struct nand_bbt_descr * *td*)
[GENERIC] scan the device for a specific bad block table

Parameters

struct mtd_info * mtd MTD device structure

uint8_t * buf temporary buffer

struct nand_bbt_descr * td descriptor for the bad block table

Description

Read the bad block table by searching for a given ident pattern. Search is preformed either from the beginning up or from the end of the device downwards. The search starts always at the start of a block. If the option NAND_BBT_PERCHIP is given, each chip is searched for a bbt, which contains the bad block information of this chip. This is necessary to provide support for certain DOC devices.

The bbt ident pattern resides in the oob area of the first page in a block.

void **search_read_bbts**(struct mtd_info * *mtd*, uint8_t * *buf*, struct nand_bbt_descr * *td*, struct nand_bbt_descr * *md*)
[GENERIC] scan the device for bad block table(s)

Parameters

struct mtd_info * mtd MTD device structure

uint8_t * buf temporary buffer

struct nand_bbt_descr * td descriptor for the bad block table

struct nand_bbt_descr * md descriptor for the bad block table mirror

Description

Search and read the bad block table(s).

int **get_bbt_block**(struct nand_chip * *this*, struct nand_bbt_descr * *td*, struct nand_bbt_descr * *md*, int *chip*)
Get the first valid eraseblock suitable to store a BBT

Parameters

struct nand_chip * this the NAND device

struct nand_bbt_descr * td the BBT description

struct nand_bbt_descr * md the mirror BBT descriptor

int chip the CHIP selector

Description

This functions returns a positive block number pointing a valid eraseblock suitable to store a BBT (i.e. in the range reserved for BBT), or -ENOSPC if all blocks are already used of marked bad. If td->pages[chip] was already pointing to a valid block we re-use it, otherwise we search for the next valid one.

void **mark_bbt_block_bad**(struct *nand_chip* * *this*, struct nand_bbt_descr * *td*, int *chip*, int *block*)
Mark one of the block reserved for BBT bad

Parameters

struct nand_chip * this the NAND device

struct nand_bbt_descr * td the BBT description

int chip the CHIP selector

int block the BBT block to mark

Description

Blocks reserved for BBT can become bad. This functions is an helper to mark such blocks as bad. It takes care of updating the in-memory BBT, marking the block as bad using a bad block marker and invalidating the associated td->pages[] entry.

int **write_bbt**(struct mtd_info * *mtd*, uint8_t * *buf*, struct nand_bbt_descr * *td*, struct nand_bbt_descr * *md*, int *chipsel*)
[GENERIC] (Re)write the bad block table

Parameters

struct mtd_info * mtd MTD device structure

uint8_t * buf temporary buffer

struct nand_bbt_descr * td descriptor for the bad block table

struct nand_bbt_descr * md descriptor for the bad block table mirror

int chipsel selector for a specific chip, -1 for all

Description

(Re)write the bad block table.

int **nand_memory_bbt**(struct mtd_info * *mtd*, struct nand_bbt_descr * *bd*)
[GENERIC] create a memory based bad block table

Parameters

struct mtd_info * mtd MTD device structure

struct nand_bbt_descr * bd descriptor for the good/bad block search pattern

Description

The function creates a memory based bbt by scanning the device for manufacturer / software marked good / bad blocks.

int **check_create**(struct mtd_info * *mtd*, uint8_t * *buf*, struct nand_bbt_descr * *bd*)
[GENERIC] create and write bbt(s) if necessary

Parameters

struct mtd_info * mtd MTD device structure

uint8_t * buf temporary buffer

struct nand_bbt_descr * bd descriptor for the good/bad block search pattern

Description

The function checks the results of the previous call to read_bbt and creates / updates the bbt(s) if necessary. Creation is necessary if no bbt was found for the chip/device. Update is necessary if one of the tables is missing or the version nr. of one table is less than the other.

void **mark_bbt_region**(struct mtd_info * *mtd*, struct nand_bbt_descr * *td*)
[GENERIC] mark the bad block table regions

Parameters

struct mtd_info * mtd MTD device structure

struct nand_bbt_descr * td bad block table descriptor

Description

The bad block table regions are marked as "bad" to prevent accidental erasures / writes. The regions are identified by the mark 0x02.

void **verify_bbt_descr**(struct mtd_info * *mtd*, struct nand_bbt_descr * *bd*)
verify the bad block description

Parameters

struct mtd_info * mtd MTD device structure

struct nand_bbt_descr * bd the table to verify

Description

This functions performs a few sanity checks on the bad block description table.

int **nand_scan_bbt**(struct mtd_info * *mtd*, struct nand_bbt_descr * *bd*)
[NAND Interface] scan, find, read and maybe create bad block table(s)

Parameters

struct mtd_info * mtd MTD device structure

struct nand_bbt_descr * bd descriptor for the good/bad block search pattern

Description

The function checks, if a bad block table(s) is/are already available. If not it scans the device for manufacturer marked good / bad blocks and writes the bad block table(s) to the selected place.

The bad block table memory is allocated here. It must be freed by calling the nand_free_bbt function.

int **nand_update_bbt**(struct mtd_info * *mtd*, loff_t *offs*)
update bad block table(s)

Parameters

struct mtd_info * mtd MTD device structure

loff_t offs the offset of the newly marked block

Description

The function updates the bad block table(s).

int **nand_create_badblock_pattern**(struct *nand_chip* * *this*)
[INTERN] Creates a BBT descriptor structure

Parameters

struct nand_chip * this NAND chip to create descriptor for

Description

This function allocates and initializes a nand_bbt_descr for BBM detection based on the properties of **this**. The new descriptor is stored in this->badblock_pattern. Thus, this->badblock_pattern should be NULL when passed to this function.

int **nand_default_bbt**(struct mtd_info * *mtd*)
[NAND Interface] Select a default bad block table for the device

Parameters

struct mtd_info * mtd MTD device structure

Description

This function selects the default bad block table support for the device and calls the nand_scan_bbt function.

int **nand_isreserved_bbt**(struct mtd_info * *mtd*, loff_t *offs*)
    [NAND Interface] Check if a block is reserved

Parameters

struct mtd_info * mtd MTD device structure

loff_t offs offset in the device

int **nand_isbad_bbt**(struct mtd_info * *mtd*, loff_t *offs*, int *allowbbt*)
    [NAND Interface] Check if a block is bad

Parameters

struct mtd_info * mtd MTD device structure

loff_t offs offset in the device

int allowbbt allow access to bad block table region

int **nand_markbad_bbt**(struct mtd_info * *mtd*, loff_t *offs*)
    [NAND Interface] Mark a block bad in the BBT

Parameters

struct mtd_info * mtd MTD device structure

loff_t offs offset of the bad block

Credits

The following people have contributed to the NAND driver:

1. Steven J. Hillsjhill@realitydiluted.com
2. David Woodhousedwmw2@infradead.org
3. Thomas Gleixnertglx@linutronix.de

A lot of users have provided bugfixes, improvements and helping hands for testing. Thanks a lot.

The following people have contributed to this document:

1. Thomas Gleixnertglx@linutronix.de

CHAPTER

TWENTYTHREE

PARALLEL PORT DEVICES

int **parport_yield**(struct pardevice * *dev*)
relinquish a parallel port temporarily

Parameters

struct pardevice * 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.

int **parport_yield_blocking**(struct pardevice * *dev*)
relinquish a parallel port temporarily

Parameters

struct pardevice * 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()*.

int **parport_wait_event**(struct parport * *port*, signed long *timeout*)
wait for an event on a parallel port

Parameters

struct parport * port port to wait on

signed long 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.

int **parport_wait_peripheral**(struct parport * *port*, unsigned char *mask*, unsigned char *result*)
wait for status lines to change in 35ms

Parameters

struct parport * port port to watch

unsigned char mask status lines to watch

unsigned char 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.

int **parport_negotiate**(struct parport * *port*, int *mode*)
negotiate an IEEE 1284 mode

Parameters

struct parport * port port to use

int 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.

ssize_t **parport_write**(struct parport * *port*, const void * *buffer*, size_t *len*)
write a block of data to a parallel port

Parameters

struct parport * port port to write to

const void * buffer data buffer (in kernel space)

size_t 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.

ssize_t **parport_read**(struct parport * *port*, void * *buffer*, size_t *len*)
read a block of data from a parallel port

Parameters

struct parport * port port to read from

void * buffer data buffer (in kernel space)

size_t 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.

long **parport_set_timeout**(struct pardevice * *dev*, long *inactivity*)
set the inactivity timeout for a device

Parameters

struct pardevice * dev device on a port

long 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.

int **__parport_register_driver**(struct parport_driver * *drv*, struct module * *owner*, const char * *mod_name*)
register a parallel port device driver

Parameters

struct parport_driver * drv structure describing the driver

struct module * owner owner module of drv

const char * 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 succeeds. but the new device model can fail and will return the error code.

void **parport_unregister_driver**(struct parport_driver * *drv*)
deregister a parallel port device driver

Parameters

struct parport_driver * 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.

struct parport * **`parport_get_port`**(struct parport * *port*)
increment a port's reference count

Parameters

`struct parport * port` the port

Description

This ensures that a struct parport pointer remains valid until the matching *`parport_put_port()`* call.

void **`parport_put_port`**(struct parport * *port*)
decrement a port's reference count

Parameters

`struct parport * 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.

struct parport * **`parport_register_port`**(unsigned long *base*, int *irq*, int *dma*, struct parport_operations * *ops*)
register a parallel port

Parameters

`unsigned long base` base I/O address

`int irq` IRQ line

`int dma` DMA channel

`struct parport_operations * 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`.

void **`parport_announce_port`**(struct parport * *port*)
tell device drivers about a parallel port

Parameters

`struct parport * 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.

void **parport_remove_port**(struct parport * *port*)
deregister a parallel port

Parameters

struct parport * 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.

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*)
register a device on a parallel port

Parameters

struct parport * port port to which the device is attached

const char * name a name to refer to the device

int (*)(void *) pf preemption callback

void (*)(void *) kf kick callback (wake-up)

void (*)(void *) irq_func interrupt handler

int flags registration flags

void * handle 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.

void **`parport_unregister_device`**(struct pardevice * *dev*)
deregister a device on a parallel port

Parameters

`struct pardevice * dev` pointer to structure representing device

Description

This undoes the effect of *`parport_register_device()`*.

struct parport * **`parport_find_number`**(int *number*)
find a parallel port by number

Parameters

`int 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()`*.

struct parport * **`parport_find_base`**(unsigned long *base*)
find a parallel port by base address

Parameters

`unsigned long 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()`*.

int **`parport_claim`**(struct pardevice * *dev*)
claim access to a parallel port device

Parameters

`struct pardevice * 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.

int **parport_claim_or_block**(struct pardevice * *dev*)
claim access to a parallel port device

Parameters

struct pardevice * 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.

void **parport_release**(struct pardevice * *dev*)
give up access to a parallel port device

Parameters

struct pardevice * 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.

struct pardevice * **parport_open**(int *devnum*, const char * *name*)
find a device by canonical device number

Parameters

int devnum canonical device number

const char * 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()*.

void **parport_close**(struct pardevice * *dev*)
close a device opened with *parport_open()*

Parameters

struct pardevice * dev device to close

Description

This is to *parport_open()* as *parport_unregister_device()* is to *parport_register_device()*.

CHAPTER

TWENTYFOUR

16X50 UART DRIVER

void **uart_update_timeout**(struct uart_port * *port*, unsigned int *cflag*, unsigned int *baud*)
update per-port FIFO timeout.

Parameters

struct uart_port * port uart_port structure describing the port

unsigned int cflag termios cflag value

unsigned int baud speed of the port

Description

Set the port FIFO timeout value. The **cflag** value should reflect the actual hardware settings.

unsigned int **uart_get_baud_rate**(struct uart_port * *port*, struct ktermios * *termios*, struct ktermios * *old*, unsigned int *min*, unsigned int *max*)
return baud rate for a particular port

Parameters

struct uart_port * port uart_port structure describing the port in question.

struct ktermios * termios desired termios settings.

struct ktermios * old old termios (or NULL)

unsigned int min minimum acceptable baud rate

unsigned int max 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").

unsigned int **uart_get_divisor**(struct uart_port * *port*, unsigned int *baud*)
return uart clock divisor

Parameters

struct uart_port * port uart_port structure describing the port.

unsigned int baud desired baud rate

Description

Calculate the uart clock divisor for the port.

void **uart_console_write**(struct uart_port * *port*, const char * *s*, unsigned int *count*, void (*putchar) (struct uart_port *, int))
write a console message to a serial port

Parameters

struct uart_port * port the port to write the message

const char * s array of characters

unsigned int count number of characters in string to write

void (*)(struct uart_port *, int) putchar function to write character to port

int **uart_parse_earlycon**(char * *p*, unsigned char * *iotype*, resource_size_t * *addr*, char ** *options*)
Parse earlycon options

Parameters

char * p ptr to 2nd field (ie., just beyond '<name>,')

unsigned char * iotype ptr for decoded iotype (out)

resource_size_t * addr ptr for decoded mapbase/iobase (out)

char ** options ptr for <options> field; NULL if not present (out)

Description

Decodes earlycon kernel command line parameters of the form
earlycon=<name>,io|mmio|mmio16|mmio32|mmio32be|mmio32native,<addr>,<options>
console=<name>,io|mmio|mmio16|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

void **uart_parse_options**(const char * *options*, int * *baud*, int * *parity*, int * *bits*, int * *flow*)
Parse serial port baud/parity/bits/flow control.

Parameters

const char * options pointer to option string

int * baud pointer to an 'int' variable for the baud rate.

int * parity pointer to an 'int' variable for the parity.

int * bits pointer to an 'int' variable for the number of data bits.

int * 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

int **uart_set_options**(struct uart_port * *port*, struct console * *co*, int *baud*, int *parity*, int *bits*, int *flow*)
setup the serial console parameters

Parameters

struct uart_port * port pointer to the serial ports uart_port structure

struct console * co console pointer

int baud baud rate

int parity parity character - 'n' (none), 'o' (odd), 'e' (even)

int bits number of data bits

int flow flow control character - 'r' (rts)

int **uart_register_driver**(struct uart_driver * *drv*)
register a driver with the uart core layer

Parameters

struct uart_driver * 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.

void **uart_unregister_driver**(struct uart_driver * *drv*)
remove a driver from the uart core layer

Parameters

struct uart_driver * 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)

int **uart_add_one_port**(struct uart_driver * *drv*, struct uart_port * *uport*)
attach a driver-defined port structure

Parameters

struct uart_driver * drv pointer to the uart low level driver structure for this port

struct uart_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.

int **uart_remove_one_port**(struct uart_driver * *drv*, struct uart_port * *uport*)
detach a driver defined port structure

Parameters

struct uart_driver * drv pointer to the uart low level driver structure for this port

struct uart_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.

void **uart_handle_dcd_change**(struct uart_port * *uport*, unsigned int *status*)
handle a change of carrier detect state

Parameters

struct uart_port * uport uart_port structure for the open port

unsigned int status new carrier detect status, nonzero if active

Description

Caller must hold uport->lock

void **uart_handle_cts_change**(struct uart_port * *uport*, unsigned int *status*)
handle a change of clear-to-send state

Parameters

struct uart_port * uport uart_port structure for the open port

unsigned int status new clear to send status, nonzero if active

Description

Caller must hold uport->lock

void **uart_insert_char**(struct uart_port * *port*, unsigned int *status*, unsigned int *overrun*, unsigned int *ch*, unsigned int *flag*)
push a char to the uart layer

Parameters

struct uart_port * port corresponding port

unsigned int status state of the serial port RX buffer (LSR for 8250)

unsigned int overrun mask of overrun bits in **status**

unsigned int ch character to push

unsigned int flag 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.

void **uart_get_rs485_mode**(struct *device* * *dev*, struct serial_rs485 * *rs485conf*)
retrieve rs485 properties for given uart

Parameters

struct device * dev uart device

struct serial_rs485 * rs485conf output parameter

Description

This function implements the device tree binding described in Documentation/devicetree/bindings/serial/rs485.txt.

struct uart_8250_port * **serial8250_get_port**(int *line*)
retrieve struct uart_8250_port

Parameters

int 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.

void **serial8250_suspend_port**(int *line*)
suspend one serial port

Parameters

int line serial line number

Description

Suspend one serial port.

void **`serial8250_resume_port`**(int *line*)
    resume one serial port

Parameters

`int line` serial line number

Description

    Resume one serial port.

int **`serial8250_register_8250_port`**(struct uart_8250_port * *up*)
    register a serial port

Parameters

`struct uart_8250_port * 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.

void **`serial8250_unregister_port`**(int *line*)
    remove a 16x50 serial port at runtime

Parameters

`int 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

TWENTYFIVE

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.

enum **pwm_polarity**
polarity of a PWM signal

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

struct **pwm_args**
board-dependent PWM arguments

Definition

```
struct pwm_args {
  unsigned int period;
  enum pwm_polarity polarity;
};
```

Members

period reference period

polarity reference polarity

Description

This structure describes board-dependent arguments attached to a PWM device. These arguments are usually retrieved from the PWM lookup table or device tree.

Do not confuse this with the PWM state: PWM arguments represent the initial configuration that users want to use on this PWM device rather than the current PWM hardware state.

struct **pwm_device**
PWM channel object

Definition

```
struct pwm_device {
  const char *label;
  unsigned long flags;
  unsigned int hwpwm;
  unsigned int pwm;
  struct pwm_chip *chip;
  void *chip_data;
  struct pwm_args args;
  struct pwm_state state;
};
```

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

args PWM arguments

state curent PWM channel state

void **pwm_get_state**(const struct *pwm_device* * *pwm*, struct pwm_state * *state*)
retrieve the current PWM state

Parameters

const struct pwm_device * pwm PWM device

struct pwm_state * state state to fill with the current PWM state

void **pwm_init_state**(const struct *pwm_device* * *pwm*, struct pwm_state * *state*)
prepare a new state to be applied with *pwm_apply_state()*

Parameters

const struct pwm_device * pwm PWM device

struct pwm_state * state state to fill with the prepared PWM state

Description

This functions prepares a state that can later be tweaked and applied to the PWM device with *pwm_apply_state()*. This is a convenient function that first retrieves the current PWM state and the replaces the period and polarity fields with the reference values defined in pwm->args. Once the function returns, you can adjust the ->enabled and ->duty_cycle fields according to your needs before calling *pwm_apply_state()*.

->duty_cycle is initially set to zero to avoid cases where the current ->duty_cycle value exceed the pwm_args->period one, which would trigger an error if the user calls *pwm_apply_state()* without adjusting ->duty_cycle first.

unsigned int **pwm_get_relative_duty_cycle**(const struct pwm_state * *state*, unsigned int *scale*)
Get a relative duty cycle value

Parameters

const struct pwm_state * state PWM state to extract the duty cycle from

unsigned int scale target scale of the relative duty cycle

Description

This functions converts the absolute duty cycle stored in **state** (expressed in nanosecond) into a value relative to the period.

For example if you want to get the duty_cycle expressed in percent, call:

pwm_get_state(pwm, `state`); duty = pwm_get_relative_duty_cycle(`state`, 100);

int **pwm_set_relative_duty_cycle**(struct pwm_state * *state*, unsigned int *duty_cycle*, unsigned int *scale*)

Set a relative duty cycle value

Parameters

struct pwm_state * state PWM state to fill

unsigned int duty_cycle relative duty cycle value

unsigned int scale scale in which **duty_cycle** is expressed

Description

This functions converts a relative into an absolute duty cycle (expressed in nanoseconds), and puts the result in state->duty_cycle.

For example if you want to configure a 50% duty cycle, call:

pwm_init_state(pwm, `state`); pwm_set_relative_duty_cycle(`state`, 50, 100); pwm_apply_state(pwm, `state`);

This functions returns -EINVAL if **duty_cycle** and/or **scale** are inconsistent (**scale** == 0 or **duty_cycle** > **scale**).

struct **pwm_ops**

PWM controller operations

Definition

```
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 period_ns);
  int (*set_polarity)(struct pwm_chip *chip, struct pwm_device *pwm, enum pwm_polarity polarity);
  int (*capture)(struct pwm_chip *chip, struct pwm_device *pwm, struct pwm_capture *result, unsigned lon
  int (*enable)(struct pwm_chip *chip, struct pwm_device *pwm);
  void (*disable)(struct pwm_chip *chip, struct pwm_device *pwm);
  int (*apply)(struct pwm_chip *chip, struct pwm_device *pwm, struct pwm_state *state);
  void (*get_state)(struct pwm_chip *chip, struct pwm_device *pwm, struct pwm_state *state);
#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

capture capture and report PWM signal

enable enable PWM output toggling

disable disable PWM output toggling

apply atomically apply a new PWM config. The state argument should be adjusted with the real hardware config (if the approximate the period or duty_cycle value, state should reflect it)

get_state get the current PWM state. This function is only called once per PWM device when the PWM chip is registered.

dbg_show optional routine to show contents in debugfs

owner helps prevent removal of modules exporting active PWMs

struct **pwm_chip**
abstract a PWM controller

Definition

```
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_args *args);
  unsigned int of_pwm_n_cells;
};
```

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

struct **pwm_capture**
PWM capture data

Definition

```
struct pwm_capture {
  unsigned int period;
  unsigned int duty_cycle;
};
```

Members

period period of the PWM signal (in nanoseconds)

duty_cycle duty cycle of the PWM signal (in nanoseconds)

int **pwm_config**(struct *pwm_device* * *pwm*, int *duty_ns*, int *period_ns*)
change a PWM device configuration

Parameters

struct pwm_device * pwm PWM device

int duty_ns “on” time (in nanoseconds)

int period_ns duration (in nanoseconds) of one cycle

Return

0 on success or a negative error code on failure.

int **pwm_set_polarity**(struct *pwm_device* * *pwm*, enum *pwm_polarity* *polarity*)
configure the polarity of a PWM signal

Parameters

struct pwm_device * pwm PWM device

enum pwm_polarity polarity new polarity of the PWM signal

Description

Note that the polarity cannot be configured while the PWM device is enabled.

Return

0 on success or a negative error code on failure.

int **pwm_enable**(struct *pwm_device* * *pwm*)
start a PWM output toggling

Parameters

struct pwm_device * pwm PWM device

Return

0 on success or a negative error code on failure.

void **pwm_disable**(struct *pwm_device* * *pwm*)
stop a PWM output toggling

Parameters

struct pwm_device * pwm PWM device

int **pwm_set_chip_data**(struct *pwm_device* * *pwm*, void * *data*)
set private chip data for a PWM

Parameters

struct pwm_device * pwm PWM device

void * data pointer to chip-specific data

Return

0 on success or a negative error code on failure.

void * **pwm_get_chip_data**(struct *pwm_device* * *pwm*)
get private chip data for a PWM

Parameters

struct pwm_device * pwm PWM device

Return

A pointer to the chip-private data for the PWM device.

int **pwmchip_add_with_polarity**(struct *pwm_chip* * *chip*, enum *pwm_polarity* *polarity*)
register a new PWM chip

Parameters

struct pwm_chip * chip the PWM chip to add

enum pwm_polarity 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.

Return

0 on success or a negative error code on failure.

int **pwmchip_add**(struct *pwm_chip* * *chip*)
register a new PWM chip

Parameters

struct pwm_chip * 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.

Return

0 on success or a negative error code on failure.

int **pwmchip_remove**(struct *pwm_chip* * *chip*)
remove a PWM chip

Parameters

struct pwm_chip * 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.

Return

0 on success or a negative error code on failure.

struct *pwm_device* * **pwm_request**(int *pwm*, const char * *label*)
request a PWM device

Parameters

int pwm global PWM device index

const char * label PWM device label

Description

This function is deprecated, use *pwm_get()* instead.

Return

A pointer to a PWM device or an ERR_PTR()-encoded error code on failure.

struct *pwm_device* * **pwm_request_from_chip**(struct *pwm_chip* * *chip*, unsigned int *index*, const char * *label*)
request a PWM device relative to a PWM chip

Parameters

struct pwm_chip * chip PWM chip

unsigned int index per-chip index of the PWM to request

const char * label a literal description string of this PWM

Return

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.

void **pwm_free**(struct *pwm_device* * *pwm*)
free a PWM device

Parameters

struct pwm_device * pwm PWM device

Description

This function is deprecated, use *pwm_put()* instead.

int **pwm_apply_state**(struct *pwm_device* * *pwm*, struct pwm_state * *state*)
atomically apply a new state to a PWM device

Parameters

struct pwm_device * pwm PWM device

struct pwm_state * state new state to apply. This can be adjusted by the PWM driver if the requested config is not achievable, for example, ->duty_cycle and ->period might be approximated.

int **pwm_capture**(struct *pwm_device* * *pwm*, struct *pwm_capture* * *result*, unsigned long *timeout*)
capture and report a PWM signal

Parameters

struct pwm_device * pwm PWM device

struct pwm_capture * result structure to fill with capture result

unsigned long timeout time to wait, in milliseconds, before giving up on capture

Return

0 on success or a negative error code on failure.

int **pwm_adjust_config**(struct *pwm_device* * *pwm*)
adjust the current PWM config to the PWM arguments

Parameters

struct pwm_device * pwm PWM device

Description

This function will adjust the PWM config to the PWM arguments provided by the DT or PWM lookup table. This is particularly useful to adapt the bootloader config to the Linux one.

struct *pwm_device* * **of_pwm_get**(struct device_node * *np*, const char * *con_id*)
request a PWM via the PWM framework

Parameters

struct device_node * np device node to get the PWM from

const char * 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.

Return

A pointer to the requested PWM device or an ERR_PTR()-encoded error code on failure.

struct *pwm_device* * **pwm_get**(struct *device* * *dev*, const char * *con_id*)
look up and request a PWM device

Parameters

struct device * dev device for PWM consumer

const char * 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.

Return

A pointer to the requested PWM device or an ERR_PTR()-encoded error code on failure.

void **pwm_put**(struct *pwm_device* * *pwm*)
    release a PWM device

Parameters

struct pwm_device * pwm PWM device

struct *pwm_device* * **devm_pwm_get**(struct *device* * *dev*, const char * *con_id*)
    resource managed *pwm_get()*

Parameters

struct device * dev device for PWM consumer

const char * con_id consumer name

Description

This function performs like *pwm_get()* but the acquired PWM device will automatically be released on driver detach.

Return

A pointer to the requested PWM device or an ERR_PTR()-encoded error code on failure.

struct *pwm_device* * **devm_of_pwm_get**(struct *device* * *dev*, struct device_node * *np*, const char * *con_id*)
    resource managed *of_pwm_get()*

Parameters

struct device * dev device for PWM consumer

struct device_node * np device node to get the PWM from

const char * con_id consumer name

Description

This function performs like *of_pwm_get()* but the acquired PWM device will automatically be released on driver detach.

Return

A pointer to the requested PWM device or an ERR_PTR()-encoded error code on failure.

void **devm_pwm_put**(struct *device* * *dev*, struct *pwm_device* * *pwm*)
    resource managed *pwm_put()*

Parameters

struct device * dev device for PWM consumer

struct pwm_device * 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.

CHAPTER

TWENTYSIX

W1: DALLAS' 1-WIRE BUS

Author David Fries

W1 API internal to the kernel

W1 API internal to the kernel

include/linux/w1.h

W1 kernel API functions.

struct **w1_reg_num**
broken out slave device id

Definition

```
struct w1_reg_num {
#if defined(__LITTLE_ENDIAN_BITFIELD);
  __u64 family:8,id:48, crc:8;
#elif defined(__BIG_ENDIAN_BITFIELD);
  __u64 crc:8,id:48, family:8;
#else;
#error "Please fix <asm/byteorder.h>";
#endif;
};
```

Members

family identifies the type of device

id along with family is the unique device id

crc checksum of the other bytes

crc checksum of the other bytes

id along with family is the unique device id

family identifies the type of device

struct **w1_slave**
holds a single slave device on the bus

Definition

```
struct w1_slave {
  struct module           *owner;
  unsigned char           name[W1_MAXNAMELEN];
  struct list_head        w1_slave_entry;
  struct w1_reg_num       reg_num;
  atomic_t refcnt;
  int ttl;
  unsigned long           flags;
  struct w1_master        *master;
  struct w1_family        *family;
  void *family_data;
  struct device           dev;
  struct device           *hwmon;
};
```

Members

owner Points to the one wire “wire” kernel module.

name Device id is ascii.

w1_slave_entry data for the linked list

reg_num the slave id in binary

refcnt reference count, delete when 0

ttl decrement per search this slave isn’t found, deatch at 0

flags bit flags for W1_SLAVE_ACTIVE W1_SLAVE_DETACH

master bus which this slave is on

family module for device family type

family_data pointer for use by the family module

dev kernel device identifier

hwmon pointer to hwmon device

struct **w1_bus_master**
operations available on a bus master

Definition

```
struct w1_bus_master {
  void *data;
  u8 (*read_bit)(void *);
  void (*write_bit)(void *, u8);
  u8 (*touch_bit)(void *, u8);
  u8 (*read_byte)(void *);
  void (*write_byte)(void *, u8);
  u8 (*read_block)(void *, u8 *, int);
  void (*write_block)(void *, const u8 *, int);
  u8 (*triplet)(void *, u8);
  u8 (*reset_bus)(void *);
  u8 (*set_pullup)(void *, int);
  void (*search)(void *, struct w1_master *, u8, w1_slave_found_callback);
};
```

Members

data the first parameter in all the functions below

read_bit Sample the line level **return** the level read (0 or 1)

write_bit Sets the line level

touch_bit the lowest-level function for devices that really support the 1-wire protocol. touch_bit(0) = write-0 cycle touch_bit(1) = write-1 / read cycle **return** the bit read (0 or 1)

read_byte Reads a bytes. Same as 8 touch_bit(1) calls. **return** the byte read

write_byte Writes a byte. Same as 8 touch_bit(x) calls.

read_block Same as a series of `read_byte()` calls **return** the number of bytes read

write_block Same as a series of `write_byte()` calls

triplet Combines two reads and a smart write for ROM searches **return** bit0=Id bit1=comp_id bit2=dir_taken

reset_bus long write-0 with a read for the presence pulse detection **return** -1=Error, 0=Device present, 1=No device present

set_pullup Put out a strong pull-up pulse of the specified duration. **return** -1=Error, 0=completed

search Really nice hardware can handles the different types of ROM search w1_master* is passed to the slave found callback. u8 is search_type, W1_SEARCH or W1_ALARM_SEARCH

Note

read_bit and write_bit are very low level functions and should only be used with hardware that doesn't really support 1-wire operations, like a parallel/serial port. Either define read_bit and write_bit OR define, at minimum, touch_bit and reset_bus.

enum **w1_master_flags**
bitfields used in w1_master.flags

Constants

W1_ABORT_SEARCH abort searching early on shutdown

W1_WARN_MAX_COUNT limit warning when the maximum count is reached

struct **w1_master**
one per bus master

Definition

```
struct w1_master {
  struct list_head        w1_master_entry;
  struct module           *owner;
  unsigned char           name[W1_MAXNAMELEN];
  struct mutex            list_mutex;
  struct list_head        slist;
  struct list_head        async_list;
  int max_slave_count, slave_count;
  unsigned long           attempts;
  int slave_ttl;
  int initialized;
  u32 id;
  int search_count;
  u64 search_id;
  atomic_t refcnt;
  void *priv;
  int enable_pullup;
  int pullup_duration;
  long flags;
  struct task_struct      *thread;
  struct mutex            mutex;
  struct mutex            bus_mutex;
  struct device_driver    *driver;
  struct device           dev;
  struct w1_bus_master    *bus_master;
  u32 seq;
};
```

Members

w1_master_entry master linked list

owner module owner

name dynamically allocate bus name

list_mutex protect slist and async_list

slist linked list of slaves

async_list linked list of netlink commands to execute

max_slave_count maximum number of slaves to search for at a time

slave_count current number of slaves known

attempts number of searches ran

slave_ttl number of searches before a slave is timed out

initialized prevent init/removal race conditions

id w1 bus number

search_count number of automatic searches to run, -1 unlimited

search_id allows continuing a search

refcnt reference count

priv private data storage

enable_pullup allows a strong pullup

pullup_duration time for the next strong pullup

flags one of w1_master_flags

thread thread for bus search and netlink commands

mutex protect most of w1_master

bus_mutex pretect concurrent bus access

driver sysfs driver

dev sysfs device

bus_master io operations available

seq sequence number used for netlink broadcasts

struct **w1_family_ops**
operations for a family type

Definition

```
struct w1_family_ops {
  int (*add_slave)(struct w1_slave *sl);
  void (*remove_slave)(struct w1_slave *sl);
  const struct attribute_group **groups;
  const struct hwmon_chip_info *chip_info;
};
```

Members

add_slave add_slave

remove_slave remove_slave

groups sysfs group

chip_info pointer to struct hwmon_chip_info

struct **w1_family**
reference counted family structure.

Definition

```
struct w1_family {
  struct list_head        family_entry;
  u8 fid;
  struct w1_family_ops    *fops;
  atomic_t refcnt;
};
```

Members

`family_entry` family linked list

`fid` 8 bit family identifier

`fops` operations for this family

`refcnt` reference counter

`module_w1_family`(*__w1_family*)
Helper macro for registering a 1-Wire families

Parameters

`__w1_family` w1_family struct

Description

Helper macro for 1-Wire families 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()`

drivers/w1/w1.c

W1 core functions.

void **`w1_search`**(struct *w1_master* * *dev*, u8 *search_type*, w1_slave_found_callback *cb*)
Performs a ROM Search & registers any devices found.

Parameters

`struct w1_master * dev` The master device to search

`u8 search_type` W1_SEARCH to search all devices, or W1_ALARM_SEARCH to return only devices in the alarmed state

`w1_slave_found_callback cb` Function to call when a device is found

Description

The 1-wire search is a simple binary tree search. For each bit of the address, we read two bits and write one bit. The bit written will put to sleep all devies that don't match that bit. When the two reads differ, the direction choice is obvious. When both bits are 0, we must choose a path to take. When we can scan all 64 bits without having to choose a path, we are done.

See "Application note 187 1-wire search algorithm" at www.maxim-ic.com

int **`w1_process_callbacks`**(struct *w1_master* * *dev*)
execute each dev->async_list callback entry

Parameters

`struct w1_master * dev` w1_master device

Description

The w1 master list_mutex must be held.

Return

1 if there were commands to executed 0 otherwise

drivers/w1/w1_family.c

Allows registering device family operations.

int **w1_register_family**(struct *w1_family* * *newf*)
register a device family driver

Parameters

struct w1_family * newf family to register

void **w1_unregister_family**(struct *w1_family* * *fent*)
unregister a device family driver

Parameters

struct w1_family * fent family to unregister

drivers/w1/w1_internal.h

W1 internal initialization for master devices.

struct **w1_async_cmd**
execute callback from the w1_process kthread

Definition

```
struct w1_async_cmd {
  struct list_head        async_entry;
  void (*cb)(struct w1_master *dev, struct w1_async_cmd *async_cmd);
};
```

Members

async_entry link entry

cb callback function, must list_del and destroy this list before returning

Description

When inserted into the w1_master async_list, w1_process will execute the callback. Embed this into the structure with the command details.

drivers/w1/w1_int.c

W1 internal initialization for master devices.

int **w1_add_master_device**(struct *w1_bus_master* * *master*)
registers a new master device

Parameters

struct w1_bus_master * master master bus device to register

void **w1_remove_master_device**(struct *w1_bus_master* * *bm*)
unregister a master device

Parameters

struct w1_bus_master * bm master bus device to remove

drivers/w1/w1_netlink.h

W1 external netlink API structures and commands.

enum **w1_cn_msg_flags**
bitfield flags for struct cn_msg.flags

Constants

W1_CN_BUNDLE Request bundling replies into fewer messagse. Be prepared to handle multiple struct cn_msg, struct w1_netlink_msg, and struct w1_netlink_cmd in one packet.

enum **w1_netlink_message_types**
message type

Constants

W1_SLAVE_ADD notification that a slave device was added

W1_SLAVE_REMOVE notification that a slave device was removed

W1_MASTER_ADD notification that a new bus master was added

W1_MASTER_REMOVE notification that a bus masterwas removed

W1_MASTER_CMD initiate operations on a specific master

W1_SLAVE_CMD sends reset, selects the slave, then does a read/write/touch operation

W1_LIST_MASTERS used to determine the bus master identifiers

struct **w1_netlink_msg**
holds w1 message type, id, and result

Definition

```
struct w1_netlink_msg {
  __u8 type;
  __u8 status;
  __u16 len;
  union {
    __u8 id[8];
    struct w1_mst {
      __u32 id;
      __u32 res;
    } mst;
  } id;
  __u8 data[0];
};
```

Members

type one of enum w1_netlink_message_types

status kernel feedback for success 0 or errno failure value

len length of data following w1_netlink_msg

id union holding bus master id (msg.id) and slave device id (id[8]).

id.id Slave ID (8 bytes)

id.mst bus master identification

id.mst.id bus master ID

id.mst.res bus master reserved

data start address of any following data

Description

The base message structure for w1 messages over netlink. The netlink connector data sequence is, struct nlmsghdr, struct cn_msg, then one or more struct w1_netlink_msg (each with optional data).

enum **w1_commands**
commands available for master or slave operations

Constants

W1_CMD_READ read len bytes

W1_CMD_WRITE write len bytes

W1_CMD_SEARCH initiate a standard search, returns only the slave devices found during that search

W1_CMD_ALARM_SEARCH search for devices that are currently alarming

W1_CMD_TOUCH Touches a series of bytes.

W1_CMD_RESET sends a bus reset on the given master

W1_CMD_SLAVE_ADD adds a slave to the given master, 8 byte slave id at data[0]

W1_CMD_SLAVE_REMOVE removes a slave to the given master, 8 byte slave id at data[0]

W1_CMD_LIST_SLAVES list of slaves registered on this master

W1_CMD_MAX number of available commands

struct **w1_netlink_cmd**
holds the command and data

Definition

```
struct w1_netlink_cmd {
  __u8 cmd;
  __u8 res;
  __u16 len;
  __u8 data[0];
};
```

Members

cmd one of enum w1_commands

res reserved

len length of data following w1_netlink_cmd

data start address of any following data

Description

One or more struct w1_netlink_cmd is placed starting at w1_netlink_msg.data each with optional data.

drivers/w1/w1_io.c

W1 input/output.

u8 **w1_touch_bit**(struct *w1_master* * *dev*, int *bit*)
Generates a write-0 or write-1 cycle and samples the level.

Parameters

struct w1_master * dev the master device

int bit 0 - write a 0, 1 - write a 0 read the level

void **w1_write_8**(struct *w1_master* * *dev*, u8 *byte*)
Writes 8 bits.

Parameters

struct w1_master * dev the master device

u8 byte the byte to write

u8 **w1_triplet**(struct *w1_master* * *dev*, int *bdir*)

- Does a triplet - used for searching ROM addresses.

Parameters

struct w1_master * dev the master device

int bdir the bit to write if both id_bit and comp_bit are 0

Description

Return bits: bit 0 = id_bit bit 1 = comp_bit bit 2 = dir_taken

If both bits 0 & 1 are set, the search should be restarted.

Return

bit fields - see above

u8 **w1_read_8**(struct *w1_master* * *dev*)
Reads 8 bits.

Parameters

struct w1_master * dev the master device

Return

the byte read

void **w1_write_block**(struct *w1_master* * *dev*, const u8 * *buf*, int *len*)
Writes a series of bytes.

Parameters

struct w1_master * dev the master device

const u8 * buf pointer to the data to write

int len the number of bytes to write

void **w1_touch_block**(struct *w1_master* * *dev*, u8 * *buf*, int *len*)
Touches a series of bytes.

Parameters

struct w1_master * dev the master device

u8 * buf pointer to the data to write

int len the number of bytes to write

u8 **w1_read_block**(struct *w1_master* * *dev*, u8 * *buf*, int *len*)
Reads a series of bytes.

Parameters

struct w1_master * dev the master device

u8 * buf pointer to the buffer to fill

int len the number of bytes to read

Return

the number of bytes read

int **w1_reset_bus**(struct *w1_master* * *dev*)
Issues a reset bus sequence.

Parameters

struct w1_master * dev the master device

Return

0=Device present, 1=No device present or error

int **w1_reset_select_slave**(struct *w1_slave* * *sl*)
reset and select a slave

Parameters

struct w1_slave * sl the slave to select

Description

Resets the bus and then selects the slave by sending either a skip rom or a rom match. A skip rom is issued if there is only one device registered on the bus. The w1 master lock must be held.

Return

0=success, anything else=error

int **w1_reset_resume_command**(struct *w1_master* * *dev*)
resume instead of another match ROM

Parameters

struct w1_master * dev the master device

Description

When the workflow with a slave amongst many requires several successive commands a reset between each, this function is similar to doing a reset then a match ROM for the last matched ROM. The advantage being that the matched ROM step is skipped in favor of the resume command. The slave must support the command of course.

If the bus has only one slave, traditionnaly the match ROM is skipped and a "SKIP ROM" is done for efficiency. On multi-slave busses, this doesn't work of course, but the resume command is the next best thing.

The w1 master lock must be held.

void **w1_next_pullup**(struct *w1_master* * *dev*, int *delay*)
register for a strong pullup

Parameters

struct w1_master * dev the master device

int delay time in milliseconds

Description

Put out a strong pull-up of the specified duration after the next write operation. Not all hardware supports strong pullups. Hardware that doesn't support strong pullups will sleep for the given time after the write operation without a strong pullup. This is a one shot request for the next write, specifying zero will clear a previous request. The w1 master lock must be held.

Return

0=success, anything else=error

void **w1_write_bit**(struct *w1_master* * *dev*, int *bit*)
Generates a write-0 or write-1 cycle.

Parameters

struct w1_master * dev the master device

int bit bit to write

Description

Only call if dev->bus_master->touch_bit is NULL

void **w1_pre_write**(struct *w1_master* * *dev*)
pre-write operations

Parameters

struct w1_master * dev the master device

Description

Pre-write operation, currently only supporting strong pullups. Program the hardware for a strong pullup, if one has been requested and the hardware supports it.

void **w1_post_write**(struct *w1_master* * *dev*)
post-write options

Parameters

struct w1_master * dev the master device

Description

Post-write operation, currently only supporting strong pullups. If a strong pullup was requested, clear it if the hardware supports them, or execute the delay otherwise, in either case clear the request.

u8 **w1_read_bit**(struct *w1_master* * *dev*)
Generates a write-1 cycle and samples the level.

Parameters

struct w1_master * dev the master device

Description

Only call if dev->bus_master->touch_bit is NULL

CHAPTER

TWENTYSEVEN

RAPIDIO SUBSYSTEM GUIDE

Author Matt Porter

Introduction

RapidIO is a high speed switched fabric interconnect with features aimed at the embedded market. RapidIO provides support for memory-mapped I/O as well as message-based transactions over the switched fabric network. RapidIO has a standardized discovery mechanism not unlike the PCI bus standard that allows simple detection of devices in a network.

This documentation is provided for developers intending to support RapidIO on new architectures, write new drivers, or to understand the subsystem internals.

Known Bugs and Limitations

Bugs

None. ;)

Limitations

1. Access/management of RapidIO memory regions is not supported
2. Multiple host enumeration is not supported

RapidIO driver interface

Drivers are provided a set of calls in order to interface with the subsystem to gather info on devices, request/map memory region resources, and manage mailboxes/doorbells.

Functions

int **rio_local_read_config_32**(struct *rio_mport* * *port*, u32 *offset*, u32 * *data*)
Read 32 bits from local configuration space

Parameters

struct rio_mport * port Master port

u32 offset Offset into local configuration space

u32 * data Pointer to read data into

Description

Reads 32 bits of data from the specified offset within the local device's configuration space.

int **rio_local_write_config_32**(struct *rio_mport* * *port*, u32 *offset*, u32 *data*)
Write 32 bits to local configuration space

Parameters

struct rio_mport * port Master port

u32 offset Offset into local configuration space

u32 data Data to be written

Description

Writes 32 bits of data to the specified offset within the local device's configuration space.

int **rio_local_read_config_16**(struct *rio_mport* * *port*, u32 *offset*, u16 * *data*)
Read 16 bits from local configuration space

Parameters

struct rio_mport * port Master port

u32 offset Offset into local configuration space

u16 * data Pointer to read data into

Description

Reads 16 bits of data from the specified offset within the local device's configuration space.

int **rio_local_write_config_16**(struct *rio_mport* * *port*, u32 *offset*, u16 *data*)
Write 16 bits to local configuration space

Parameters

struct rio_mport * port Master port

u32 offset Offset into local configuration space

u16 data Data to be written

Description

Writes 16 bits of data to the specified offset within the local device's configuration space.

int **rio_local_read_config_8**(struct *rio_mport* * *port*, u32 *offset*, u8 * *data*)
Read 8 bits from local configuration space

Parameters

struct rio_mport * port Master port

u32 offset Offset into local configuration space

u8 * data Pointer to read data into

Description

Reads 8 bits of data from the specified offset within the local device's configuration space.

int **rio_local_write_config_8**(struct *rio_mport* * *port*, u32 *offset*, u8 *data*)
Write 8 bits to local configuration space

Parameters

struct rio_mport * port Master port

u32 offset Offset into local configuration space

u8 data Data to be written

Description

Writes 8 bits of data to the specified offset within the local device's configuration space.

int **rio_read_config_32**(struct *rio_dev* * *rdev*, u32 *offset*, u32 * *data*)
    Read 32 bits from configuration space

Parameters

struct rio_dev * rdev RIO device

u32 offset Offset into device configuration space

u32 * data Pointer to read data into

Description

Reads 32 bits of data from the specified offset within the RIO device's configuration space.

int **rio_write_config_32**(struct *rio_dev* * *rdev*, u32 *offset*, u32 *data*)
    Write 32 bits to configuration space

Parameters

struct rio_dev * rdev RIO device

u32 offset Offset into device configuration space

u32 data Data to be written

Description

Writes 32 bits of data to the specified offset within the RIO device's configuration space.

int **rio_read_config_16**(struct *rio_dev* * *rdev*, u32 *offset*, u16 * *data*)
    Read 16 bits from configuration space

Parameters

struct rio_dev * rdev RIO device

u32 offset Offset into device configuration space

u16 * data Pointer to read data into

Description

Reads 16 bits of data from the specified offset within the RIO device's configuration space.

int **rio_write_config_16**(struct *rio_dev* * *rdev*, u32 *offset*, u16 *data*)
    Write 16 bits to configuration space

Parameters

struct rio_dev * rdev RIO device

u32 offset Offset into device configuration space

u16 data Data to be written

Description

Writes 16 bits of data to the specified offset within the RIO device's configuration space.

int **rio_read_config_8**(struct *rio_dev* * *rdev*, u32 *offset*, u8 * *data*)
    Read 8 bits from configuration space

Parameters

struct rio_dev * rdev RIO device

u32 offset Offset into device configuration space

u8 * data Pointer to read data into

Description

Reads 8 bits of data from the specified offset within the RIO device's configuration space.

int **rio_write_config_8**(struct *rio_dev* * *rdev*, u32 *offset*, u8 *data*)
Write 8 bits to configuration space

Parameters

struct rio_dev * rdev RIO device

u32 offset Offset into device configuration space

u8 data Data to be written

Description

Writes 8 bits of data to the specified offset within the RIO device's configuration space.

int **rio_send_doorbell**(struct *rio_dev* * *rdev*, u16 *data*)
Send a doorbell message to a device

Parameters

struct rio_dev * rdev RIO device

u16 data Doorbell message data

Description

Send a doorbell message to a RIO device. The doorbell message has a 16-bit info field provided by the **data** argument.

void **rio_init_mbox_res**(struct resource * *res*, int *start*, int *end*)
Initialize a RIO mailbox resource

Parameters

struct resource * res resource struct

int start start of mailbox range

int end end of mailbox range

Description

This function is used to initialize the fields of a resource for use as a mailbox resource. It initializes a range of mailboxes using the start and end arguments.

void **rio_init_dbell_res**(struct resource * *res*, u16 *start*, u16 *end*)
Initialize a RIO doorbell resource

Parameters

struct resource * res resource struct

u16 start start of doorbell range

u16 end end of doorbell range

Description

This function is used to initialize the fields of a resource for use as a doorbell resource. It initializes a range of doorbell messages using the start and end arguments.

RIO_DEVICE(*dev*, *ven*)
macro used to describe a specific RIO device

Parameters

dev the 16 bit RIO device ID

ven the 16 bit RIO vendor ID

Description

This macro is used to create a struct rio_device_id that matches a specific device. The assembly vendor and assembly device fields will be set to `RIO_ANY_ID`.

int **rio_add_outb_message**(struct *rio_mport* * *mport*, struct *rio_dev* * *rdev*, int *mbox*, void * *buffer*, size_t *len*)
Add RIO message to an outbound mailbox queue

Parameters

struct rio_mport * mport RIO master port containing the outbound queue

struct rio_dev * rdev RIO device the message is be sent to

int mbox The outbound mailbox queue

void * buffer Pointer to the message buffer

size_t len Length of the message buffer

Description

Adds a RIO message buffer to an outbound mailbox queue for transmission. Returns 0 on success.

int **rio_add_inb_buffer**(struct *rio_mport* * *mport*, int *mbox*, void * *buffer*)
Add buffer to an inbound mailbox queue

Parameters

struct rio_mport * mport Master port containing the inbound mailbox

int mbox The inbound mailbox number

void * buffer Pointer to the message buffer

Description

Adds a buffer to an inbound mailbox queue for reception. Returns 0 on success.

void * **rio_get_inb_message**(struct *rio_mport* * *mport*, int *mbox*)
Get A RIO message from an inbound mailbox queue

Parameters

struct rio_mport * mport Master port containing the inbound mailbox

int mbox The inbound mailbox number

Description

Get a RIO message from an inbound mailbox queue. Returns 0 on success.

const char * **rio_name**(struct *rio_dev* * *rdev*)
Get the unique RIO device identifier

Parameters

struct rio_dev * rdev RIO device

Description

Get the unique RIO device identifier. Returns the device identifier string.

void * **rio_get_drvdata**(struct *rio_dev* * *rdev*)
Get RIO driver specific data

Parameters

struct rio_dev * rdev RIO device

Description

Get RIO driver specific data. Returns a pointer to the driver specific data.

void **rio_set_drvdata**(struct *rio_dev* * *rdev*, void * *data*)
Set RIO driver specific data

Parameters

struct rio_dev * rdev RIO device

void * data Pointer to driver specific data

Description

Set RIO driver specific data. device struct driver data pointer is set to the **data** argument.

struct *rio_dev* * **rio_dev_get**(struct *rio_dev* * *rdev*)
Increments the reference count of the RIO device structure

Parameters

struct rio_dev * rdev RIO device being referenced

Description

Each live reference to a device should be refcounted.

Drivers for RIO devices should normally record such references in their `probe()` methods, when they bind to a device, and release them by calling *rio_dev_put()*, in their `disconnect()` methods.

void **rio_dev_put**(struct *rio_dev* * *rdev*)
Release a use of the RIO device structure

Parameters

struct rio_dev * rdev RIO device being disconnected

Description

Must be called when a user of a device is finished with it. When the last user of the device calls this function, the memory of the device is freed.

int **rio_register_driver**(struct *rio_driver* * *rdrv*)
register a new RIO driver

Parameters

struct rio_driver * rdrv the RIO driver structure to register

Description

Adds a *struct rio_driver* to the list of registered drivers. Returns a negative value on error, otherwise 0. If no error occurred, the driver remains registered even if no device was claimed during registration.

void **rio_unregister_driver**(struct *rio_driver* * *rdrv*)
unregister a RIO driver

Parameters

struct rio_driver * rdrv the RIO driver structure to unregister

Description

Deletes the *struct rio_driver* from the list of registered RIO drivers, gives it a chance to clean up by calling its `remove()` function for each device it was responsible for, and marks those devices as driverless.

u16 **rio_local_get_device_id**(struct *rio_mport* * *port*)
Get the base/extended device id for a port

Parameters

struct rio_mport * port RIO master port from which to get the deviceid

Description

Reads the base/extended device id from the local device implementing the master port. Returns the 8/16-bit device id.

int **rio_query_mport**(struct *rio_mport* * *port*, struct *rio_mport_attr* * *mport_attr*)
  Query mport device attributes

Parameters

struct rio_mport * port mport device to query

struct rio_mport_attr * mport_attr mport attributes data structure

Description

Returns attributes of specified mport through the pointer to attributes data structure.

struct *rio_net* * **rio_alloc_net**(struct *rio_mport* * *mport*)
  Allocate and initialize a new RIO network data structure

Parameters

struct rio_mport * mport Master port associated with the RIO network

Description

Allocates a RIO network structure, initializes per-network list heads, and adds the associated master port to the network list of associated master ports. Returns a RIO network pointer on success or NULL on failure.

void **rio_local_set_device_id**(struct *rio_mport* * *port*, u16 *did*)
  Set the base/extended device id for a port

Parameters

struct rio_mport * port RIO master port

u16 did Device ID value to be written

Description

Writes the base/extended device id from a device.

int **rio_add_device**(struct *rio_dev* * *rdev*)
  Adds a RIO device to the device model

Parameters

struct rio_dev * rdev RIO device

Description

Adds the RIO device to the global device list and adds the RIO device to the RIO device list. Creates the generic sysfs nodes for an RIO device.

int **rio_request_inb_mbox**(struct *rio_mport* * *mport*, void * *dev_id*, int *mbox*, int *entries*, void (*minb) (struct *rio_mport* * *mport*, void **dev_id*, int *mbox*, int *slot*)
  request inbound mailbox service

Parameters

struct rio_mport * mport RIO master port from which to allocate the mailbox resource

void * dev_id Device specific pointer to pass on event

int mbox Mailbox number to claim

int entries Number of entries in inbound mailbox queue

void (*) (struct rio_mport * mport, void *dev_id, int mbox, int slot) minb Callback to execute when inbound message is received

Description

Requests ownership of an inbound mailbox resource and binds a callback function to the resource. Returns `0` on success.

int **rio_release_inb_mbox**(struct *rio_mport* * *mport*, int *mbox*)
release inbound mailbox message service

Parameters

struct rio_mport * mport RIO master port from which to release the mailbox resource

int mbox Mailbox number to release

Description

Releases ownership of an inbound mailbox resource. Returns 0 if the request has been satisfied.

int **rio_request_outb_mbox**(struct *rio_mport* * *mport*, void * *dev_id*, int *mbox*, int *entries*, void (*moutb) (struct *rio_mport* * *mport*, void **dev_id*, int *mbox*, int *slot*)
request outbound mailbox service

Parameters

struct rio_mport * mport RIO master port from which to allocate the mailbox resource

void * dev_id Device specific pointer to pass on event

int mbox Mailbox number to claim

int entries Number of entries in outbound mailbox queue

void (*) (struct rio_mport * mport, void *dev_id, int mbox, int slot) moutb Callback to execute when outbound message is sent

Description

Requests ownership of an outbound mailbox resource and binds a callback function to the resource. Returns 0 on success.

int **rio_release_outb_mbox**(struct *rio_mport* * *mport*, int *mbox*)
release outbound mailbox message service

Parameters

struct rio_mport * mport RIO master port from which to release the mailbox resource

int mbox Mailbox number to release

Description

Releases ownership of an inbound mailbox resource. Returns 0 if the request has been satisfied.

int **rio_request_inb_dbell**(struct *rio_mport* * *mport*, void * *dev_id*, u16 *start*, u16 *end*, void (*dinb) (struct *rio_mport* * *mport*, void **dev_id*, u16 *src*, u16 *dst*, u16 *info*)
request inbound doorbell message service

Parameters

struct rio_mport * mport RIO master port from which to allocate the doorbell resource

void * dev_id Device specific pointer to pass on event

u16 start Doorbell info range start

u16 end Doorbell info range end

void (*) (struct rio_mport * mport, void *dev_id, u16 src, u16 dst, u16 info) dinb Callback to execute when doorbell is received

Description

Requests ownership of an inbound doorbell resource and binds a callback function to the resource. Returns 0 if the request has been satisfied.

int **rio_release_inb_dbell**(struct *rio_mport* * *mport*, u16 *start*, u16 *end*)
release inbound doorbell message service

Parameters

struct rio_mport * mport RIO master port from which to release the doorbell resource

u16 start Doorbell info range start

u16 end Doorbell info range end

Description

Releases ownership of an inbound doorbell resource and removes callback from the doorbell event list. Returns 0 if the request has been satisfied.

struct resource * **rio_request_outb_dbell**(struct *rio_dev* * *rdev*, u16 *start*, u16 *end*)
request outbound doorbell message range

Parameters

struct rio_dev * rdev RIO device from which to allocate the doorbell resource

u16 start Doorbell message range start

u16 end Doorbell message range end

Description

Requests ownership of a doorbell message range. Returns a resource if the request has been satisfied or NULL on failure.

int **rio_release_outb_dbell**(struct *rio_dev* * *rdev*, struct resource * *res*)
release outbound doorbell message range

Parameters

struct rio_dev * rdev RIO device from which to release the doorbell resource

struct resource * res Doorbell resource to be freed

Description

Releases ownership of a doorbell message range. Returns 0 if the request has been satisfied.

int **rio_add_mport_pw_handler**(struct *rio_mport* * *mport*, void * *context*, int (*pwcback) (struct *rio_mport* **mport*, void **context*, union rio_pw_msg **msg*, int *step*)
add port-write message handler into the list of mport specific pw handlers

Parameters

struct rio_mport * mport RIO master port to bind the portwrite callback

void * context Handler specific context to pass on event

int (*)(struct rio_mport *mport, void *context, union rio_pw_msg *msg, int step) pwcback
Callback to execute when portwrite is received

Description

Returns 0 if the request has been satisfied.

int **rio_del_mport_pw_handler**(struct *rio_mport* * *mport*, void * *context*, int (*pwcback) (struct *rio_mport* **mport*, void **context*, union rio_pw_msg **msg*, int *step*)
remove port-write message handler from the list of mport specific pw handlers

Parameters

struct rio_mport * mport RIO master port to bind the portwrite callback

void * context Registered handler specific context to pass on event

int (*)(struct rio_mport *mport, void *context, union rio_pw_msg *msg, int step) pwcback
Registered callback function

Description

Returns 0 if the request has been satisfied.

int **rio_request_inb_pwrite**(struct *rio_dev* * *rdev*, int (*pwcback) (struct *rio_dev* **rdev*, union rio_pw_msg **msg*, int *step*)
request inbound port-write message service for specific RapidIO device

Parameters

struct rio_dev * rdev RIO device to which register inbound port-write callback routine

int (*)(struct rio_dev *rdev, union rio_pw_msg *msg, int step) pwcback Callback routine to execute when port-write is received

Description

Binds a port-write callback function to the RapidIO device. Returns 0 if the request has been satisfied.

int **rio_release_inb_pwrite**(struct *rio_dev* * *rdev*)
release inbound port-write message service associated with specific RapidIO device

Parameters

struct rio_dev * rdev RIO device which registered for inbound port-write callback

Description

Removes callback from the rio_dev structure. Returns 0 if the request has been satisfied.

void **rio_pw_enable**(struct *rio_mport* * *mport*, int *enable*)
Enables/disables port-write handling by a master port

Parameters

struct rio_mport * mport Master port associated with port-write handling

int enable 1=enable, 0=disable

int **rio_map_inb_region**(struct *rio_mport* * *mport*, dma_addr_t *local*, u64 *rbase*, u32 *size*, u32 *rflags*)

- Map inbound memory region.

Parameters

struct rio_mport * mport Master port.

dma_addr_t local physical address of memory region to be mapped

u64 rbase RIO base address assigned to this window

u32 size Size of the memory region

u32 rflags Flags for mapping.

Return

0 – Success.

This function will create the mapping from RIO space to local memory.

void **rio_unmap_inb_region**(struct *rio_mport* * *mport*, dma_addr_t *lstart*)

- Unmap the inbound memory region

Parameters

struct rio_mport * mport Master port

dma_addr_t lstart physical address of memory region to be unmapped

int **rio_map_outb_region**(struct *rio_mport* * *mport*, u16 *destid*, u64 *rbase*, u32 *size*, u32 *rflags*, dma_addr_t * *local*)

- Map outbound memory region.

Parameters

struct rio_mport * mport Master port.

u16 destid destination id window points to

u64 rbase RIO base address window translates to

u32 size Size of the memory region

u32 rflags Flags for mapping.

dma_addr_t * local physical address of memory region mapped

Return

0 – Success.

This function will create the mapping from RIO space to local memory.

void **rio_unmap_outb_region**(struct *rio_mport* * *mport*, u16 *destid*, u64 *rstart*)

- Unmap the inbound memory region

Parameters

struct rio_mport * mport Master port

u16 destid destination id mapping points to

u64 rstart RIO base address window translates to

u32 **rio_mport_get_physefb**(struct *rio_mport* * *port*, int *local*, u16 *destid*, u8 *hopcount*, u32 * *rmap*)

Helper function that returns register offset for Physical Layer Extended Features Block.

Parameters

struct rio_mport * port Master port to issue transaction

int local Indicate a local master port or remote device access

u16 destid Destination ID of the device

u8 hopcount Number of switch hops to the device

u32 * rmap pointer to location to store register map type info

struct *rio_dev* * **rio_get_comptag**(u32 *comp_tag*, struct *rio_dev* * *from*)

Begin or continue searching for a RIO device by component tag

Parameters

u32 comp_tag RIO component tag to match

struct rio_dev * from Previous RIO device found in search, or NULL for new search

Description

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

int **rio_set_port_lockout**(struct *rio_dev* * *rdev*, u32 *pnum*, int *lock*)

Sets/clears LOCKOUT bit (RIO EM 1.3) for a switch port.

Parameters

struct rio_dev * rdev Pointer to RIO device control structure

u32 pnum Switch port number to set LOCKOUT bit

int lock Operation : set (=1) or clear (=0)

int **rio_enable_rx_tx_port**(struct *rio_mport* * *port*, int *local*, u16 *destid*, u8 *hopcount*, u8 *port_num*)
enable input receiver and output transmitter of given port

Parameters

struct rio_mport * port Master port associated with the RIO network

int local local=1 select local port otherwise a far device is reached

u16 destid Destination ID of the device to check host bit

u8 hopcount Number of hops to reach the target

u8 port_num Port (-number on switch) to enable on a far end device

Description

Returns 0 or 1 from on General Control Command and Status Register (EXT_PTR+0x3C)

int **rio_mport_chk_dev_access**(struct *rio_mport* * *mport*, u16 *destid*, u8 *hopcount*)
Validate access to the specified device.

Parameters

struct rio_mport * mport Master port to send transactions

u16 destid Device destination ID in network

u8 hopcount Number of hops into the network

int **rio_inb_pwrite_handler**(struct *rio_mport* * *mport*, union rio_pw_msg * *pw_msg*)
inbound port-write message handler

Parameters

struct rio_mport * mport mport device associated with port-write

union rio_pw_msg * pw_msg pointer to inbound port-write message

Description

Processes an inbound port-write message. Returns 0 if the request has been satisfied.

u32 **rio_mport_get_efb**(struct *rio_mport* * *port*, int *local*, u16 *destid*, u8 *hopcount*, u32 *from*)
get pointer to next extended features block

Parameters

struct rio_mport * port Master port to issue transaction

int local Indicate a local master port or remote device access

u16 destid Destination ID of the device

u8 hopcount Number of switch hops to the device

u32 from Offset of current Extended Feature block header (if 0 starts from ExtFeaturePtr)

u32 **rio_mport_get_feature**(struct *rio_mport* * *port*, int *local*, u16 *destid*, u8 *hopcount*, int *ftr*)
query for devices' extended features

Parameters

struct rio_mport * port Master port to issue transaction

int local Indicate a local master port or remote device access

u16 destid Destination ID of the device

u8 hopcount Number of switch hops to the device

int ftr Extended feature code

Description

Tell if a device supports a given RapidIO capability. Returns the offset of the requested extended feature block within the device's RIO configuration space or 0 in case the device does not support it.

struct *rio_dev* * **rio_get_asm**(u16 *vid*, u16 *did*, u16 *asm_vid*, u16 *asm_did*, struct *rio_dev* * *from*)
Begin or continue searching for a RIO device by vid/did/asm_vid/asm_did

Parameters

u16 vid RIO vid to match or RIO_ANY_ID to match all vids

u16 did RIO did to match or RIO_ANY_ID to match all dids

u16 asm_vid RIO asm_vid to match or RIO_ANY_ID to match all asm_vids

u16 asm_did RIO asm_did to match or RIO_ANY_ID to match all asm_dids

struct rio_dev * from Previous RIO device found in search, or NULL for new search

Description

Iterates through the list of known RIO devices. If a RIO device is found with a matching **vid**, **did**, **asm_vid**, **asm_did**, the reference count to the device is incrememted and a pointer to its device structure is returned. Otherwise, NULL is returned. A new search is initiated by passing NULL to the **from** argument. Otherwise, if **from** is not NULL, searches continue from next device on the global list. The reference count for **from** is always decremented if it is not NULL.

struct *rio_dev* * **rio_get_device**(u16 *vid*, u16 *did*, struct *rio_dev* * *from*)
Begin or continue searching for a RIO device by vid/did

Parameters

u16 vid RIO vid to match or RIO_ANY_ID to match all vids

u16 did RIO did to match or RIO_ANY_ID to match all dids

struct rio_dev * from Previous RIO device found in search, or NULL for new search

Description

Iterates through the list of known RIO devices. If a RIO device is found with a matching **vid** and **did**, the reference count to the device is incrememted and a pointer to its device structure is returned. Otherwise, NULL is returned. A new search is initiated by passing NULL to the **from** argument. Otherwise, if **from** is not NULL, searches continue from next device on the global list. The reference count for **from** is always decremented if it is not NULL.

int **rio_lock_device**(struct *rio_mport* * *port*, u16 *destid*, u8 *hopcount*, int *wait_ms*)
Acquires host device lock for specified device

Parameters

struct rio_mport * port Master port to send transaction

u16 destid Destination ID for device/switch

u8 hopcount Hopcount to reach switch

int wait_ms Max wait time in msec (0 = no timeout)

Description

Attepts to acquire host device lock for specified device Returns 0 if device lock acquired or EINVAL if timeout expires.

int **rio_unlock_device**(struct *rio_mport* * *port*, u16 *destid*, u8 *hopcount*)
Releases host device lock for specified device

Parameters

struct rio_mport * port Master port to send transaction

u16 destid Destination ID for device/switch

u8 hopcount Hopcount to reach switch

Description

Returns 0 if device lock released or EINVAL if fails.

int **rio_route_add_entry**(struct *rio_dev* * *rdev*, u16 *table*, u16 *route_destid*, u8 *route_port*, int *lock*)
Add a route entry to a switch routing table

Parameters

struct rio_dev * rdev RIO device

u16 table Routing table ID

u16 route_destid Destination ID to be routed

u8 route_port Port number to be routed

int lock apply a hardware lock on switch device flag (1=lock, 0=no_lock)

Description

If available calls the switch specific `add_entry()` method to add a route entry into a switch routing table. Otherwise uses standard RT update method as defined by RapidIO specification. A specific routing table can be selected using the **table** argument if a switch has per port routing tables or the standard (or global) table may be used by passing `RIO_GLOBAL_TABLE` in **table**.

Returns `0` on success or `-EINVAL` on failure.

int **rio_route_get_entry**(struct *rio_dev* * *rdev*, u16 *table*, u16 *route_destid*, u8 * *route_port*, int *lock*)
Read an entry from a switch routing table

Parameters

struct rio_dev * rdev RIO device

u16 table Routing table ID

u16 route_destid Destination ID to be routed

u8 * route_port Pointer to read port number into

int lock apply a hardware lock on switch device flag (1=lock, 0=no_lock)

Description

If available calls the switch specific `get_entry()` method to fetch a route entry from a switch routing table. Otherwise uses standard RT read method as defined by RapidIO specification. A specific routing table can be selected using the **table** argument if a switch has per port routing tables or the standard (or global) table may be used by passing `RIO_GLOBAL_TABLE` in **table**.

Returns `0` on success or `-EINVAL` on failure.

int **rio_route_clr_table**(struct *rio_dev* * *rdev*, u16 *table*, int *lock*)
Clear a switch routing table

Parameters

struct rio_dev * rdev RIO device

u16 table Routing table ID

int lock apply a hardware lock on switch device flag (1=lock, 0=no_lock)

Description

If available calls the switch specific `clr_table()` method to clear a switch routing table. Otherwise uses standard RT write method as defined by RapidIO specification. A specific routing table can be selected using the **table** argument if a switch has per port routing tables or the standard (or global) table may be used by passing `RIO_GLOBAL_TABLE` in **table**.

Returns `0` on success or `-EINVAL` on failure.

struct dma_chan * **`rio_request_mport_dma`**(struct *rio_mport* * *mport*)
request RapidIO capable DMA channel associated with specified local RapidIO mport device.

Parameters

`struct rio_mport * mport` RIO mport to perform DMA data transfers

Description

Returns pointer to allocated DMA channel or NULL if failed.

struct dma_chan * **`rio_request_dma`**(struct *rio_dev* * *rdev*)
request RapidIO capable DMA channel that supports specified target RapidIO device.

Parameters

`struct rio_dev * rdev` RIO device associated with DMA transfer

Description

Returns pointer to allocated DMA channel or NULL if failed.

void **`rio_release_dma`**(struct dma_chan * *dchan*)
release specified DMA channel

Parameters

`struct dma_chan * dchan` DMA channel to release

struct dma_async_tx_descriptor * **`rio_dma_prep_xfer`**(struct dma_chan * *dchan*, u16 *destid*, struct rio_dma_data * *data*, enum dma_transfer_direction *direction*, unsigned long *flags*)
RapidIO specific wrapper for device_prep_slave_sg callback defined by DMAENGINE.

Parameters

`struct dma_chan * dchan` DMA channel to configure

`u16 destid` target RapidIO device destination ID

`struct rio_dma_data * data` RIO specific data descriptor

`enum dma_transfer_direction direction` DMA data transfer direction (TO or FROM the device)

`unsigned long flags` dmaengine defined flags

Description

Initializes RapidIO capable DMA channel for the specified data transfer. Uses DMA channel private extension to pass information related to remote target RIO device.

Return

pointer to DMA transaction descriptor if successful, error-valued pointer or NULL if failed.

struct dma_async_tx_descriptor * **`rio_dma_prep_slave_sg`**(struct *rio_dev* * *rdev*, struct dma_chan * *dchan*, struct rio_dma_data * *data*, enum dma_transfer_direction *direction*, unsigned long *flags*)
RapidIO specific wrapper for device_prep_slave_sg callback defined by DMAENGINE.

Parameters

`struct rio_dev * rdev` RIO device control structure

`struct dma_chan * dchan` DMA channel to configure

`struct rio_dma_data * data` RIO specific data descriptor

`enum dma_transfer_direction direction` DMA data transfer direction (TO or FROM the device)

unsigned long flags dmaengine defined flags

Description

Initializes RapidIO capable DMA channel for the specified data transfer. Uses DMA channel private extension to pass information related to remote target RIO device.

Return

pointer to DMA transaction descriptor if successful, error-valued pointer or NULL if failed.

int **rio_register_scan**(int *mport_id*, struct *rio_scan* * *scan_ops*)
enumeration/discovery method registration interface

Parameters

int mport_id mport device ID for which fabric scan routine has to be set (RIO_MPORT_ANY = set for all available mports)

struct rio_scan * scan_ops enumeration/discovery operations structure

Description

Registers enumeration/discovery operations with RapidIO subsystem and attaches it to the specified mport device (or all available mports if RIO_MPORT_ANY is specified).

Returns error if the mport already has an enumerator attached to it. In case of RIO_MPORT_ANY skips mports with valid scan routines (no error).

int **rio_unregister_scan**(int *mport_id*, struct *rio_scan* * *scan_ops*)
removes enumeration/discovery method from mport

Parameters

int mport_id mport device ID for which fabric scan routine has to be unregistered (RIO_MPORT_ANY = apply to all mports that use the specified scan_ops)

struct rio_scan * scan_ops enumeration/discovery operations structure

Description

Removes enumeration or discovery method assigned to the specified mport device. If RIO_MPORT_ANY is specified, removes the specified operations from all mports that have them attached.

Internals

This chapter contains the autogenerated documentation of the RapidIO subsystem.

Structures

struct **rio_switch**
RIO switch info

Definition

```
struct rio_switch {
  struct list_head node;
  u8 *route_table;
  u32 port_ok;
  struct rio_switch_ops *ops;
  spinlock_t lock;
  struct rio_dev *nextdev[0];
};
```

Members

node Node in global list of switches

route_table Copy of switch routing table

port_ok Status of each port (one bit per port) - OK=1 or UNINIT=0

ops pointer to switch-specific operations

lock lock to serialize operations updates

nextdev Array of per-port pointers to the next attached device

struct **rio_switch_ops**
Per-switch operations

Definition

```
struct rio_switch_ops {
  struct module *owner;
  int (*add_entry) (struct rio_mport *mport, u16 destid, u8 hopcount, u16 table, u16 route_destid, u8 ro
  int (*get_entry) (struct rio_mport *mport, u16 destid, u8 hopcount, u16 table, u16 route_destid, u8 *r
  int (*clr_table) (struct rio_mport *mport, u16 destid, u8 hopcount, u16 table);
  int (*set_domain) (struct rio_mport *mport, u16 destid, u8 hopcount, u8 sw_domain);
  int (*get_domain) (struct rio_mport *mport, u16 destid, u8 hopcount, u8 *sw_domain);
  int (*em_init) (struct rio_dev *dev);
  int (*em_handle) (struct rio_dev *dev, u8 swport);
};
```

Members

owner The module owner of this structure

add_entry Callback for switch-specific route add function

get_entry Callback for switch-specific route get function

clr_table Callback for switch-specific clear route table function

set_domain Callback for switch-specific domain setting function

get_domain Callback for switch-specific domain get function

em_init Callback for switch-specific error management init function

em_handle Callback for switch-specific error management handler function

Description

Defines the operations that are necessary to initialize/control a particular RIO switch device.

struct **rio_dev**
RIO device info

Definition

```
struct rio_dev {
  struct list_head global_list;
  struct list_head net_list;
  struct rio_net *net;
  bool do_enum;
  u16 did;
  u16 vid;
  u32 device_rev;
  u16 asm_did;
  u16 asm_vid;
  u16 asm_rev;
  u16 efptr;
  u32 pef;
  u32 swpinfo;
  u32 src_ops;
  u32 dst_ops;
  u32 comp_tag;
  u32 phys_efptr;
  u32 phys_rmap;
  u32 em_efptr;
  u64 dma_mask;
  struct rio_driver *driver;
  struct device dev;
  struct resource riores[RIO_MAX_DEV_RESOURCES];
  int (*pwcback) (struct rio_dev *rdev, union rio_pw_msg *msg, int step);
  u16 destid;
  u8 hopcount;
  struct rio_dev *prev;
  atomic_t state;
  struct rio_switch rswitch[0];
};
```

Members

global_list Node in list of all RIO devices

net_list Node in list of RIO devices in a network

net Network this device is a part of

do_enum Enumeration flag

did Device ID

vid Vendor ID

device_rev Device revision

asm_did Assembly device ID

asm_vid Assembly vendor ID

asm_rev Assembly revision

efptr Extended feature pointer

pef Processing element features

swpinfo Switch port info

src_ops Source operation capabilities

dst_ops Destination operation capabilities

comp_tag RIO component tag

phys_efptr RIO device extended features pointer

phys_rmap LP-Serial Register Map Type (1 or 2)

em_efptr RIO Error Management features pointer

dma_mask Mask of bits of RIO address this device implements

driver Driver claiming this device

dev Device model device

riores RIO resources this device owns

pwcback port-write callback function for this device

destid Network destination ID (or associated destid for switch)

hopcount Hopcount to this device

prev Previous RIO device connected to the current one

state device state

rswitch struct rio_switch (if valid for this device)

struct **rio_msg**
RIO message event

Definition

```
struct rio_msg {
  struct resource *res;
  void (*mcback) (struct rio_mport * mport, void *dev_id, int mbox, int slot);
};
```

Members

res Mailbox resource

mcback Message event callback

struct **rio_dbell**
RIO doorbell event

Definition

```
struct rio_dbell {
  struct list_head node;
  struct resource *res;
  void (*dinb) (struct rio_mport *mport, void *dev_id, u16 src, u16 dst, u16 info);
  void *dev_id;
};
```

Members

node Node in list of doorbell events

res Doorbell resource

dinb Doorbell event callback

dev_id Device specific pointer to pass on event

struct **rio_mport**
RIO master port info

Definition

```
struct rio_mport {
  struct list_head dbells;
  struct list_head pwrites;
  struct list_head node;
  struct list_head nnode;
  struct rio_net *net;
  struct mutex lock;
  struct resource iores;
  struct resource riores[RIO_MAX_MPORT_RESOURCES];
  struct rio_msg inb_msg[RIO_MAX_MBOX];
  struct rio_msg outb_msg[RIO_MAX_MBOX];
  int host_deviceid;
  struct rio_ops *ops;
  unsigned char id;
  unsigned char index;
  unsigned int sys_size;
  u32 phys_efptr;
  u32 phys_rmap;
  unsigned char name[RIO_MAX_MPORT_NAME];
  struct device dev;
  void *priv;
#ifdef CONFIG_RAPIDIO_DMA_ENGINE;
  struct dma_device       dma;
#endif;
  struct rio_scan *nscan;
  atomic_t state;
  unsigned int pwe_refcnt;
};
```

Members

dbells List of doorbell events

pwrites List of portwrite events

node Node in global list of master ports

nnode Node in network list of master ports

net RIO net this mport is attached to

lock lock to synchronize lists manipulations

iores I/O mem resource that this master port interface owns

riores RIO resources that this master port interfaces owns

inb_msg RIO inbound message event descriptors

outb_msg RIO outbound message event descriptors

host_deviceid Host device ID associated with this master port

ops configuration space functions

id Port ID, unique among all ports

index Port index, unique among all port interfaces of the same type

sys_size RapidIO common transport system size

phys_efptr RIO port extended features pointer

phys_rmap LP-Serial EFB Register Mapping type (1 or 2).

name Port name string

dev device structure associated with an mport

priv Master port private data

dma DMA device associated with mport

nscan RapidIO network enumeration/discovery operations

state mport device state

pwe_refcnt port-write enable ref counter to track enable/disable requests

struct **rio_net**
RIO network info

Definition

```
struct rio_net {
  struct list_head node;
  struct list_head devices;
  struct list_head switches;
  struct list_head mports;
  struct rio_mport *hport;
  unsigned char id;
  struct device dev;
  void *enum_data;
  void (*release)(struct rio_net *net);
};
```

Members

node Node in global list of RIO networks

devices List of devices in this network

switches List of switches in this network

mports List of master ports accessing this network

hport Default port for accessing this network

id RIO network ID

dev Device object

enum_data private data specific to a network enumerator

release enumerator-specific release callback

struct **rio_mport_attr**
RIO mport device attributes

Definition

```
struct rio_mport_attr {
  int flags;
  int link_speed;
  int link_width;
  int dma_max_sge;
  int dma_max_size;
  int dma_align;
};
```

Members

flags mport device capability flags

link_speed SRIO link speed value (as defined by RapidIO specification)

link_width SRIO link width value (as defined by RapidIO specification)

dma_max_sge number of SG list entries that can be handled by DMA channel(s)

dma_max_size max number of bytes in single DMA transfer (SG entry)

dma_align alignment shift for DMA operations (as for other DMA operations)

struct **rio_ops**
Low-level RIO configuration space operations

Definition

```
struct rio_ops {
  int (*lcread) (struct rio_mport *mport, int index, u32 offset, int len, u32 *data);
  int (*lcwrite) (struct rio_mport *mport, int index, u32 offset, int len, u32 data);
  int (*cread) (struct rio_mport *mport, int index, u16 destid, u8 hopcount, u32 offset, int len, u32 *d
  int (*cwrite) (struct rio_mport *mport, int index, u16 destid, u8 hopcount, u32 offset, int len, u32 d
  int (*dsend) (struct rio_mport *mport, int index, u16 destid, u16 data);
  int (*pwenable) (struct rio_mport *mport, int enable);
  int (*open_outb_mbox)(struct rio_mport *mport, void *dev_id, int mbox, int entries);
  void (*close_outb_mbox)(struct rio_mport *mport, int mbox);
  int (*open_inb_mbox)(struct rio_mport *mport, void *dev_id, int mbox, int entries);
  void (*close_inb_mbox)(struct rio_mport *mport, int mbox);
  int (*add_outb_message)(struct rio_mport *mport, struct rio_dev *rdev, int mbox, void *buffer, size_t
  int (*add_inb_buffer)(struct rio_mport *mport, int mbox, void *buf);
  void *(*get_inb_message)(struct rio_mport *mport, int mbox);
  int (*map_inb)(struct rio_mport *mport, dma_addr_t lstart, u64 rstart, u64 size, u32 flags);
  void (*unmap_inb)(struct rio_mport *mport, dma_addr_t lstart);
  int (*query_mport)(struct rio_mport *mport, struct rio_mport_attr *attr);
  int (*map_outb)(struct rio_mport *mport, u16 destid, u64 rstart, u32 size, u32 flags, dma_addr_t *ladd
  void (*unmap_outb)(struct rio_mport *mport, u16 destid, u64 rstart);
};
```

Members

lcread Callback to perform local (master port) read of config space.

lcwrite Callback to perform local (master port) write of config space.

cread Callback to perform network read of config space.

cwrite Callback to perform network write of config space.

dsend Callback to send a doorbell message.

pwenable Callback to enable/disable port-write message handling.

open_outb_mbox Callback to initialize outbound mailbox.

close_outb_mbox Callback to shut down outbound mailbox.

open_inb_mbox Callback to initialize inbound mailbox.

close_inb_mbox Callback to shut down inbound mailbox.

add_outb_message Callback to add a message to an outbound mailbox queue.

add_inb_buffer Callback to add a buffer to an inbound mailbox queue.

get_inb_message Callback to get a message from an inbound mailbox queue.

map_inb Callback to map RapidIO address region into local memory space.

unmap_inb Callback to unmap RapidIO address region mapped with `map_inb()`.

query_mport Callback to query mport device attributes.

map_outb Callback to map outbound address region into local memory space.

unmap_outb Callback to unmap outbound RapidIO address region.

struct **rio_driver**
    RIO driver info

Definition

```
struct rio_driver {
  struct list_head node;
  char *name;
  const struct rio_device_id *id_table;
  int (*probe) (struct rio_dev * dev, const struct rio_device_id * id);
  void (*remove) (struct rio_dev * dev);
  void (*shutdown)(struct rio_dev *dev);
  int (*suspend) (struct rio_dev * dev, u32 state);
  int (*resume) (struct rio_dev * dev);
  int (*enable_wake) (struct rio_dev * dev, u32 state, int enable);
  struct device_driver driver;
};
```

Members

node Node in list of drivers

name RIO driver name

id_table RIO device ids to be associated with this driver

probe RIO device inserted

remove RIO device removed

shutdown shutdown notification callback

suspend RIO device suspended

resume RIO device awakened

enable_wake RIO device enable wake event

driver LDM driver struct

Description

Provides info on a RIO device driver for insertion/removal and power management purposes.

struct **rio_scan**
RIO enumeration and discovery operations

Definition

```
struct rio_scan {
  struct module *owner;
  int (*enumerate)(struct rio_mport *mport, u32 flags);
  int (*discover)(struct rio_mport *mport, u32 flags);
};
```

Members

owner The module owner of this structure

enumerate Callback to perform RapidIO fabric enumeration.

discover Callback to perform RapidIO fabric discovery.

struct **rio_scan_node**
list node to register RapidIO enumeration and discovery methods with RapidIO core.

Definition

```
struct rio_scan_node {
  int mport_id;
  struct list_head node;
  struct rio_scan *ops;
};
```

Members

mport_id ID of an mport (net) serviced by this enumerator

node node in global list of registered enumerators

ops RIO enumeration and discovery operations

Enumeration and Discovery

u16 **rio_destid_alloc**(struct *rio_net* * *net*)
Allocate next available destID for given network

Parameters

struct rio_net * net RIO network

Description

Returns next available device destination ID for the specified RIO network. Marks allocated ID as one in use. Returns RIO_INVALID_DESTID if new destID is not available.

int **rio_destid_reserve**(struct *rio_net* * *net*, u16 *destid*)
Reserve the specivied destID

Parameters

struct rio_net * net RIO network

u16 destid destID to reserve

Description

Tries to reserve the specified destID. Returns 0 if successful.

void **rio_destid_free**(struct *rio_net* * *net*, u16 *destid*)
free a previously allocated destID

Parameters

struct rio_net * net RIO network

u16 destid destID to free

Description

Makes the specified destID available for use.

u16 **rio_destid_first**(struct *rio_net* * *net*)
return first destID in use

Parameters

struct rio_net * net RIO network

u16 **rio_destid_next**(struct *rio_net* * *net*, u16 *from*)
return next destID in use

Parameters

struct rio_net * net RIO network

u16 from destination ID from which search shall continue

u16 **rio_get_device_id**(struct *rio_mport* * *port*, u16 *destid*, u8 *hopcount*)
Get the base/extended device id for a device

Parameters

struct rio_mport * port RIO master port

u16 destid Destination ID of device

u8 hopcount Hopcount to device

Description

Reads the base/extended device id from a device. Returns the 8/16-bit device ID.

void **rio_set_device_id**(struct *rio_mport* * *port*, u16 *destid*, u8 *hopcount*, u16 *did*)
Set the base/extended device id for a device

Parameters

struct rio_mport * port RIO master port

u16 destid Destination ID of device

u8 hopcount Hopcount to device

u16 did Device ID value to be written

Description

Writes the base/extended device id from a device.

int **rio_clear_locks**(struct *rio_net* * *net*)
Release all host locks and signal enumeration complete

Parameters

struct rio_net * net RIO network to run on

Description

Marks the component tag CSR on each device with the enumeration complete flag. When complete, it then release the host locks on each device. Returns 0 on success or `-EINVAL` on failure.

int **rio_enum_host**(struct *rio_mport* * *port*)
Set host lock and initialize host destination ID

Parameters

struct rio_mport * port Master port to issue transaction

Description

Sets the local host master port lock and destination ID register with the host device ID value. The host device ID value is provided by the platform. Returns `0` on success or `-1` on failure.

int **rio_device_has_destid**(struct *rio_mport* * *port*, int *src_ops*, int *dst_ops*)
Test if a device contains a destination ID register

Parameters

struct rio_mport * port Master port to issue transaction

int src_ops RIO device source operations

int dst_ops RIO device destination operations

Description

Checks the provided **src_ops** and **dst_ops** for the necessary transaction capabilities that indicate whether or not a device will implement a destination ID register. Returns 1 if true or 0 if false.

void **rio_release_dev**(struct *device* * *dev*)
Frees a RIO device struct

Parameters

struct device * dev LDM device associated with a RIO device struct

Description

Gets the RIO device struct associated a RIO device struct. The RIO device struct is freed.

int **rio_is_switch**(struct *rio_dev* * *rdev*)
Tests if a RIO device has switch capabilities

Parameters

struct rio_dev * rdev RIO device

Description

Gets the RIO device Processing Element Features register contents and tests for switch capabilities. Returns 1 if the device is a switch or 0 if it is not a switch. The RIO device struct is freed.

struct *rio_dev* * **rio_setup_device**(struct *rio_net* * *net*, struct *rio_mport* * *port*, u16 *destid*, u8 *hopcount*, int *do_enum*)
Allocates and sets up a RIO device

Parameters

struct rio_net * net RIO network

struct rio_mport * port Master port to send transactions

u16 destid Current destination ID

u8 hopcount Current hopcount

int do_enum Enumeration/Discovery mode flag

Description

Allocates a RIO device and configures fields based on configuration space contents. If device has a destination ID register, a destination ID is either assigned in enumeration mode or read from configuration space in discovery mode. If the device has switch capabilities, then a switch is allocated and configured appropriately. Returns a pointer to a RIO device on success or NULL on failure.

int **rio_sport_is_active**(struct *rio_dev* * *rdev*, int *sp*)
Tests if a switch port has an active connection.

Parameters

struct rio_dev * rdev RapidIO device object

int sp Switch port number

Description

Reads the port error status CSR for a particular switch port to determine if the port has an active link. Returns RIO_PORT_N_ERR_STS_PORT_OK if the port is active or 0 if it is inactive.

u16 **rio_get_host_deviceid_lock**(struct *rio_mport* * *port*, u8 *hopcount*)
Reads the Host Device ID Lock CSR on a device

Parameters

struct rio_mport * port Master port to send transaction

u8 hopcount Number of hops to the device

Description

Used during enumeration to read the Host Device ID Lock CSR on a RIO device. Returns the value of the lock register.

int **rio_enum_peer**(struct *rio_net* * *net*, struct *rio_mport* * *port*, u8 *hopcount*, struct *rio_dev* * *prev*, int *prev_port*)
Recursively enumerate a RIO network through a master port

Parameters

struct rio_net * net RIO network being enumerated

struct rio_mport * port Master port to send transactions

u8 hopcount Number of hops into the network

struct rio_dev * prev Previous RIO device connected to the enumerated one

int prev_port Port on previous RIO device

Description

Recursively enumerates a RIO network. Transactions are sent via the master port passed in **port**.

int **rio_enum_complete**(struct *rio_mport* * *port*)
Tests if enumeration of a network is complete

Parameters

struct rio_mport * port Master port to send transaction

Description

Tests the PGCCSR discovered bit for non-zero value (enumeration complete flag). Return 1 if enumeration is complete or 0 if enumeration is incomplete.

int **rio_disc_peer**(struct *rio_net* * *net*, struct *rio_mport* * *port*, u16 *destid*, u8 *hopcount*, struct *rio_dev* * *prev*, int *prev_port*)
Recursively discovers a RIO network through a master port

Parameters

struct rio_net * net RIO network being discovered

struct rio_mport * port Master port to send transactions

u16 destid Current destination ID in network

u8 hopcount Number of hops into the network

struct rio_dev * prev previous rio_dev

int prev_port previous port number

Description

Recursively discovers a RIO network. Transactions are sent via the master port passed in **port**.

int **rio_mport_is_active**(struct *rio_mport* * *port*)
Tests if master port link is active

Parameters

struct rio_mport * port Master port to test

Description

Reads the port error status CSR for the master port to determine if the port has an active link. Returns RIO_PORT_N_ERR_STS_PORT_OK if the master port is active or 0 if it is inactive.

void **rio_update_route_tables**(struct *rio_net* * *net*)
Updates route tables in switches

Parameters

struct rio_net * net RIO network to run update on

Description

For each enumerated device, ensure that each switch in a system has correct routing entries. Add routes for devices that where unknown dirung the first enumeration pass through the switch.

void **rio_init_em**(struct *rio_dev* * *rdev*)
Initializes RIO Error Management (for switches)

Parameters

struct rio_dev * rdev RIO device

Description

For each enumerated switch, call device-specific error management initialization routine (if supplied by the switch driver).

int **rio_enum_mport**(struct *rio_mport* * *mport*, u32 *flags*)
Start enumeration through a master port

Parameters

struct rio_mport * mport Master port to send transactions

u32 flags Enumeration control flags

Description

Starts the enumeration process. If somebody has enumerated our master port device, then give up. If not and we have an active link, then start recursive peer enumeration. Returns 0 if enumeration succeeds or -EBUSY if enumeration fails.

void **rio_build_route_tables**(struct *rio_net* * *net*)
Generate route tables from switch route entries

Parameters

struct rio_net * net RIO network to run route tables scan on

Description

For each switch device, generate a route table by copying existing route entries from the switch.

int **rio_disc_mport**(struct *rio_mport* * *mport*, u32 *flags*)
Start discovery through a master port

Parameters

struct rio_mport * mport Master port to send transactions

u32 flags discovery control flags

Description

Starts the discovery process. If we have an active link, then wait for the signal that enumeration is complete (if wait is allowed). When enumeration completion is signaled, start recursive peer discovery. Returns `0` if discovery succeeds or `-EBUSY` on failure.

int **rio_basic_attach**(void)

Parameters

void no arguments

Description

When this enumeration/discovery method is loaded as a module this function registers its specific enumeration and discover routines for all available RapidIO mport devices. The "scan" command line parameter controls ability of the module to start RapidIO enumeration/discovery automatically.

Returns 0 for success or -EIO if unable to register itself.

This enumeration/discovery method cannot be unloaded and therefore does not provide a matching cleanup_module routine.

Driver functionality

int **rio_setup_inb_dbell**(struct *rio_mport* * *mport*, void * *dev_id*, struct resource * *res*, void (*dinb) (struct *rio_mport* * *mport*, void **dev_id*, u16 *src*, u16 *dst*, u16 *info*)
bind inbound doorbell callback

Parameters

struct rio_mport * mport RIO master port to bind the doorbell callback

void * dev_id Device specific pointer to pass on event

struct resource * res Doorbell message resource

void (*) (struct rio_mport * mport, void *dev_id, u16 src, u16 dst, u16 info) dinb
Callback to execute when doorbell is received

Description

Adds a doorbell resource/callback pair into a port's doorbell event list. Returns 0 if the request has been satisfied.

int **rio_chk_dev_route**(struct *rio_dev* * *rdev*, struct *rio_dev* ** *nrdev*, int * *npnum*)
Validate route to the specified device.

Parameters

struct rio_dev * rdev RIO device failed to respond

struct rio_dev ** nrdev Last active device on the route to rdev

int * npnum nrdev's port number on the route to rdev

Description

Follows a route to the specified RIO device to determine the last available device (and corresponding RIO port) on the route.

int **rio_chk_dev_access**(struct *rio_dev* * *rdev*)
Validate access to the specified device.

Parameters

struct rio_dev * rdev Pointer to RIO device control structure

int **rio_get_input_status**(struct *rio_dev* * *rdev*, int *pnum*, u32 * *lnkresp*)
Sends a Link-Request/Input-Status control symbol and returns link-response (if requested).

Parameters

struct rio_dev * rdev RIO devive to issue Input-status command

int pnum Device port number to issue the command

u32 * lnkresp Response from a link partner

int **rio_clr_err_stopped**(struct *rio_dev* * *rdev*, u32 *pnum*, u32 *err_status*)
Clears port Error-stopped states.

Parameters

struct rio_dev * rdev Pointer to RIO device control structure

u32 pnum Switch port number to clear errors

u32 err_status port error status (if 0 reads register from device)

Description

TODO: Currently this routine is not compatible with recovery process specified for idt_gen3 RapidIO switch devices. It has to be reviewed to implement universal recovery process that is compatible full range off available devices. IDT gen3 switch driver now implements HW-specific error handler that issues soft port reset to the port to reset ERR_STOP bits and ackIDs.

int **rio_std_route_add_entry**(struct *rio_mport* * *mport*, u16 *destid*, u8 *hopcount*, u16 *table*, u16 *route_destid*, u8 *route_port*)
Add switch route table entry using standard registers defined in RIO specification rev.1.3

Parameters

struct rio_mport * mport Master port to issue transaction

u16 destid Destination ID of the device

u8 hopcount Number of switch hops to the device

u16 table routing table ID (global or port-specific)

u16 route_destid destID entry in the RT

u8 route_port destination port for specified destID

int **rio_std_route_get_entry**(struct *rio_mport* * *mport*, u16 *destid*, u8 *hopcount*, u16 *table*, u16 *route_destid*, u8 * *route_port*)
Read switch route table entry (port number) associated with specified destID using standard registers defined in RIO specification rev.1.3

Parameters

struct rio_mport * mport Master port to issue transaction

u16 destid Destination ID of the device

u8 hopcount Number of switch hops to the device

u16 table routing table ID (global or port-specific)

u16 route_destid destID entry in the RT

u8 * route_port returned destination port for specified destID

int **rio_std_route_clr_table**(struct *rio_mport* * *mport*, u16 *destid*, u8 *hopcount*, u16 *table*)
Clear swotch route table using standard registers defined in RIO specification rev.1.3.

Parameters

struct rio_mport * mport Master port to issue transaction

u16 destid Destination ID of the device

u8 hopcount Number of switch hops to the device

u16 table routing table ID (global or port-specific)

struct *rio_mport* * **rio_find_mport**(int *mport_id*)
find RIO mport by its ID

Parameters

int mport_id number (ID) of mport device

Description

Given a RIO mport number, the desired mport is located in the global list of mports. If the mport is found, a pointer to its data structure is returned. If no mport is found, NULL is returned.

int **rio_mport_scan**(int *mport_id*)
execute enumeration/discovery on the specified mport

Parameters

int mport_id number (ID) of mport device

RIO_LOP_READ(*size*, *type*, *len*)
Generate rio_local_read_config_* functions

Parameters

size Size of configuration space read (8, 16, 32 bits)

type C type of value argument

len Length of configuration space read (1, 2, 4 bytes)

Description

Generates rio_local_read_config_* functions used to access configuration space registers on the local device.

RIO_LOP_WRITE(*size*, *type*, *len*)
Generate rio_local_write_config_* functions

Parameters

size Size of configuration space write (8, 16, 32 bits)

type C type of value argument

len Length of configuration space write (1, 2, 4 bytes)

Description

Generates rio_local_write_config_* functions used to access configuration space registers on the local device.

RIO_OP_READ(*size*, *type*, *len*)
Generate rio_mport_read_config_* functions

Parameters

size Size of configuration space read (8, 16, 32 bits)

type C type of value argument

len Length of configuration space read (1, 2, 4 bytes)

Description

Generates rio_mport_read_config_* functions used to access configuration space registers on the local device.

RIO_OP_WRITE(*size*, *type*, *len*)
Generate rio_mport_write_config_* functions

Parameters

size Size of configuration space write (8, 16, 32 bits)

type C type of value argument

len Length of configuration space write (1, 2, 4 bytes)

Description

Generates rio_mport_write_config_* functions used to access configuration space registers on the local device.

Device model support

const struct *rio_device_id* * **rio_match_device**(const struct *rio_device_id* * *id*, const struct *rio_dev* * *rdev*)
Tell if a RIO device has a matching RIO device id structure

Parameters

const struct rio_device_id * id the RIO device id structure to match against

const struct rio_dev * rdev the RIO device structure to match against

Description

Used from driver probe and bus matching to check whether a RIO device matches a device id structure provided by a RIO driver. Returns the matching `struct rio_device_id` or `NULL` if there is no match.

int **rio_device_probe**(struct *device* * *dev*)
Tell if a RIO device structure has a matching RIO device id structure

Parameters

struct device * dev the RIO device structure to match against

Description

return 0 and set rio_dev->driver when drv claims rio_dev, else error

int **rio_device_remove**(struct *device* * *dev*)
Remove a RIO device from the system

Parameters

struct device * dev the RIO device structure to match against

Description

Remove a RIO device from the system. If it has an associated driver, then run the driver `remove()` method. Then update the reference count.

int **rio_match_bus**(struct *device* * *dev*, struct *device_driver* * *drv*)
Tell if a RIO device structure has a matching RIO driver device id structure

Parameters

struct device * dev the standard device structure to match against

struct device_driver * drv the standard driver structure containing the ids to match against

Description

Used by a driver to check whether a RIO device present in the system is in its list of supported devices. Returns 1 if there is a matching `struct rio_device_id` or 0 if there is no match.

int **rio_bus_init**(void)
Register the RapidIO bus with the device model

Parameters

void no arguments

Description

Registers the RIO mport device class and RIO bus type with the Linux device model.

PPC32 support

int **fsl_local_config_read**(struct *rio_mport* * *mport*, int *index*, u32 *offset*, int *len*, u32 * *data*)
Generate a MPC85xx local config space read

Parameters

struct rio_mport * mport RapidIO master port info

int index ID of RapdiIO interface

u32 offset Offset into configuration space

int len Length (in bytes) of the maintenance transaction

u32 * data Value to be read into

Description

Generates a MPC85xx local configuration space read. Returns `0` on success or `-EINVAL` on failure.

int **fsl_local_config_write**(struct *rio_mport* * *mport*, int *index*, u32 *offset*, int *len*, u32 *data*)
Generate a MPC85xx local config space write

Parameters

struct rio_mport * mport RapidIO master port info

int index ID of RapdiIO interface

u32 offset Offset into configuration space

int len Length (in bytes) of the maintenance transaction

u32 data Value to be written

Description

Generates a MPC85xx local configuration space write. Returns `0` on success or `-EINVAL` on failure.

int **fsl_rio_config_read**(struct *rio_mport* * *mport*, int *index*, u16 *destid*, u8 *hopcount*, u32 *offset*, int *len*, u32 * *val*)
Generate a MPC85xx read maintenance transaction

Parameters

struct rio_mport * mport RapidIO master port info

int index ID of RapdiIO interface

u16 destid Destination ID of transaction

u8 hopcount Number of hops to target device

u32 offset Offset into configuration space

int len Length (in bytes) of the maintenance transaction

u32 * val Location to be read into

Description

Generates a MPC85xx read maintenance transaction. Returns `0` on success or `-EINVAL` on failure.

int **fsl_rio_config_write**(struct *rio_mport* * *mport*, int *index*, u16 *destid*, u8 *hopcount*, u32 *offset*, int *len*, u32 *val*)
Generate a MPC85xx write maintenance transaction

Parameters

struct rio_mport * mport RapidIO master port info

int index ID of RapdiIO interface

u16 destid Destination ID of transaction

u8 hopcount Number of hops to target device

u32 offset Offset into configuration space

int len Length (in bytes) of the maintenance transaction

u32 val Value to be written

Description

Generates an MPC85xx write maintenance transaction. Returns `0` on success or `-EINVAL` on failure.

int **fsl_rio_setup**(struct platform_device * *dev*)
Setup Freescale PowerPC RapidIO interface

Parameters

struct platform_device * dev platform_device pointer

Description

Initializes MPC85xx RapidIO hardware interface, configures master port with system-specific info, and registers the master port with the RapidIO subsystem.

Credits

The following people have contributed to the RapidIO subsystem directly or indirectly:

1. Matt Portermporter@kernel.crashing.org
2. Randy Vinsonrvinson@mvista.com
3. Dan Malekdan@embeddedalley.com

The following people have contributed to this document:

1. Matt Portermporter@kernel.crashing.org

CHAPTER

TWENTYEIGHT

WRITING S390 CHANNEL DEVICE DRIVERS

Author Cornelia Huck

Introduction

This document describes the interfaces available for device drivers that drive s390 based channel attached I/O devices. This includes interfaces for interaction with the hardware and interfaces for interacting with the common driver core. Those interfaces are provided by the s390 common I/O layer.

The document assumes a familarity with the technical terms associated with the s390 channel I/O architecture. For a description of this architecture, please refer to the "z/Architecture: Principles of Operation", IBM publication no. SA22-7832.

While most I/O devices on a s390 system are typically driven through the channel I/O mechanism described here, there are various other methods (like the diag interface). These are out of the scope of this document.

The s390 common I/O layer also provides access to some devices that are not strictly considered I/O devices. They are considered here as well, although they are not the focus of this document.

Some additional information can also be found in the kernel source under Documentation/s390/driver-model.txt.

The css bus

The css bus contains the subchannels available on the system. They fall into several categories:

- Standard I/O subchannels, for use by the system. They have a child device on the ccw bus and are described below.
- I/O subchannels bound to the vfio-ccw driver. See Documentation/s390/vfio-ccw.txt.
- Message subchannels. No Linux driver currently exists.
- CHSC subchannels (at most one). The chsc subchannel driver can be used to send asynchronous chsc commands.
- eADM subchannels. Used for talking to storage class memory.

The ccw bus

The ccw bus typically contains the majority of devices available to a s390 system. Named after the channel command word (ccw), the basic command structure used to address its devices, the ccw bus contains so-called channel attached devices. They are addressed via I/O subchannels, visible on the css bus. A device driver for channel-attached devices, however, will never interact with the subchannel directly, but only via the I/O device on the ccw bus, the ccw device.

I/O functions for channel-attached devices

Some hardware structures have been translated into C structures for use by the common I/O layer and device drivers. For more information on the hardware structures represented here, please consult the Principles of Operation.

struct **ccw1**
channel command word

Definition

```
struct ccw1 {
  __u8 cmd_code;
  __u8 flags;
  __u16 count;
  __u32 cda;
};
```

Members

cmd_code command code

flags flags, like IDA addressing, etc.

count byte count

cda data address

Description

The ccw is the basic structure to build channel programs that perform operations with the device or the control unit. Only Format-1 channel command words are supported.

struct **ccw0**
channel command word

Definition

```
struct ccw0 {
  __u8 cmd_code;
  __u32 cda : 24;
  __u8 flags;
  __u8 reserved;
  __u16 count;
};
```

Members

cmd_code command code

cda data address

flags flags, like IDA addressing, etc.

reserved will be ignored

count byte count

Description

The format-0 ccw structure.

struct **erw**
extended report word

Definition

```
struct erw {
  __u32 res0  : 3;
  __u32 auth  : 1;
  __u32 pvrf  : 1;
  __u32 cpt   : 1;
  __u32 fsavf : 1;
  __u32 cons  : 1;
  __u32 scavf : 1;
  __u32 fsaf  : 1;
  __u32 scnt  : 6;
  __u32 res16 : 16;
};
```

Members

res0 reserved

auth authorization check

pvrf path-verification-required flag

cpt channel-path timeout

fsavf failing storage address validity flag

cons concurrent sense

scavf secondary ccw address validity flag

fsaf failing storage address format

scnt sense count, if **cons** == 1

res16 reserved

struct **erw_eadm**
EADM Subchannel extended report word

Definition

```
struct erw_eadm {
  __u32 : 16;
  __u32 b : 1;
  __u32 r : 1;
  __u32 : 14;
};
```

Members

b aob error

r arsb error

struct **sublog**
subchannel logout area

Definition

```
struct sublog {
  __u32 res0  : 1;
  __u32 esf   : 7;
  __u32 lpum  : 8;
  __u32 arep  : 1;
  __u32 fvf   : 5;
  __u32 sacc  : 2;
  __u32 termc : 2;
  __u32 devsc : 1;
  __u32 serr  : 1;
  __u32 ioerr : 1;
  __u32 seqc  : 3;
};
```

Members

res0 reserved

esf extended status flags

lpum last path used mask

arep ancillary report

fvf field-validity flags

sacc storage access code

termc termination code

devsc device-status check

serr secondary error

ioerr i/o-error alert

seqc sequence code

struct **esw0**
    Format 0 Extended Status Word (ESW)

Definition

```
struct esw0 {
  struct sublog sublog;
  struct erw erw;
  __u32 faddr[2];
  __u32 saddr;
};
```

Members

sublog subchannel logout

erw extended report word

faddr failing storage address

saddr secondary ccw address

struct **esw1**
    Format 1 Extended Status Word (ESW)

Definition

```
struct esw1 {
  __u8 zero0;
  __u8 lpum;
  __u16 zero16;
  struct erw erw;
  __u32 zeros[3];
};
```

Members

zero0 reserved zeros

lpum last path used mask

zero16 reserved zeros

erw extended report word

zeros three fullwords of zeros

struct **esw2**
Format 2 Extended Status Word (ESW)

Definition

```
struct esw2 {
  __u8 zero0;
  __u8 lpum;
  __u16 dcti;
  struct erw erw;
  __u32 zeros[3];
};
```

Members

zero0 reserved zeros

lpum last path used mask

dcti device-connect-time interval

erw extended report word

zeros three fullwords of zeros

struct **esw3**
Format 3 Extended Status Word (ESW)

Definition

```
struct esw3 {
  __u8 zero0;
  __u8 lpum;
  __u16 res;
  struct erw erw;
  __u32 zeros[3];
};
```

Members

zero0 reserved zeros

lpum last path used mask

res reserved

erw extended report word

zeros three fullwords of zeros

struct **esw_eadm**
EADM Subchannel Extended Status Word (ESW)

Definition

```
struct esw_eadm {
  __u32 sublog;
  struct erw_eadm erw;
  __u32 : 32;
  __u32 : 32;
  __u32 : 32;
};
```

Members

sublog subchannel logout

erw extended report word

struct **irb**
interruption response block

Definition

```
struct irb {
  union scsw scsw;
  union {
    struct esw0 esw0;
    struct esw1 esw1;
    struct esw2 esw2;
    struct esw3 esw3;
    struct esw_eadm eadm;
  } esw;
  __u8 ecw[32];
};
```

Members

scsw subchannel status word

esw extended status word

ecw extended control word

Description

The irb that is handed to the device driver when an interrupt occurs. For solicited interrupts, the common I/O layer already performs checks whether a field is valid; a field not being valid is always passed as 0. If a unit check occurred, **ecw** may contain sense data; this is retrieved by the common I/O layer itself if the device doesn't support concurrent sense (so that the device driver never needs to perform basic sene itself). For unsolicited interrupts, the irb is passed as-is (expect for sense data, if applicable).

struct **ciw**
command information word (CIW) layout

Definition

```
struct ciw {
  __u32 et       :  2;
  __u32 reserved :  2;
  __u32 ct       :  4;
  __u32 cmd      :  8;
  __u32 count    : 16;
};
```

Members

et entry type

reserved reserved bits

ct command type

cmd command code

count command count

struct **ccw_dev_id**
unique identifier for ccw devices

Definition

```
struct ccw_dev_id {
  u8 ssid;
  u16 devno;
};
```

Members

ssid subchannel set id

devno device number

Description

This structure is not directly based on any hardware structure. The hardware identifies a device by its device number and its subchannel, which is in turn identified by its id. In order to get a unique identifier for ccw devices across subchannel sets, **struct** ccw_dev_id has been introduced.

int **ccw_dev_id_is_equal**(struct *ccw_dev_id* * *dev_id1*, struct *ccw_dev_id* * *dev_id2*)
compare two ccw_dev_ids

Parameters

struct ccw_dev_id * dev_id1 a ccw_dev_id

struct ccw_dev_id * dev_id2 another ccw_dev_id

Return

1 if the two structures are equal field-by-field, 0 if not.

Context

any

u8 **pathmask_to_pos**(u8 *mask*)
find the position of the left-most bit in a pathmask

Parameters

u8 mask pathmask with at least one bit set

ccw devices

Devices that want to initiate channel I/O need to attach to the ccw bus. Interaction with the driver core is done via the common I/O layer, which provides the abstractions of ccw devices and ccw device drivers.

The functions that initiate or terminate channel I/O all act upon a ccw device structure. Device drivers must not bypass those functions or strange side effects may happen.

struct **ccw_device**
channel attached device

Definition

```
struct ccw_device {
  spinlock_t *ccwlock;
  struct ccw_device_id id;
  struct ccw_driver *drv;
  struct device dev;
  int online;
  void (*handler) (struct ccw_device *, unsigned long, struct irb *);
};
```

Members

ccwlock pointer to device lock

id id of this device

drv ccw driver for this device

dev embedded device structure

online online status of device

handler interrupt handler

Description

handler is a member of the device rather than the driver since a driver can have different interrupt handlers for different ccw devices (multi-subchannel drivers).

struct **ccw_driver**
device driver for channel attached devices

Definition

```
struct ccw_driver {
  struct ccw_device_id *ids;
  int (*probe) (struct ccw_device *);
  void (*remove) (struct ccw_device *);
  int (*set_online) (struct ccw_device *);
  int (*set_offline) (struct ccw_device *);
  int (*notify) (struct ccw_device *, int);
  void (*path_event) (struct ccw_device *, int *);
  void (*shutdown) (struct ccw_device *);
  int (*prepare) (struct ccw_device *);
  void (*complete) (struct ccw_device *);
  int (*freeze)(struct ccw_device *);
  int (*thaw) (struct ccw_device *);
  int (*restore)(struct ccw_device *);
  enum uc_todo (*uc_handler) (struct ccw_device *, struct irb *);
  struct device_driver driver;
  enum interruption_class int_class;
};
```

Members

ids ids supported by this driver

probe function called on probe

remove function called on remove

set_online called when setting device online

set_offline called when setting device offline

notify notify driver of device state changes

path_event notify driver of channel path events

shutdown called at device shutdown

prepare prepare for pm state transition

complete undo work done in **prepare**

freeze callback for freezing during hibernation snapshotting

thaw undo work done in **freeze**

restore callback for restoring after hibernation

uc_handler callback for unit check handler

driver embedded device driver structure

int_class interruption class to use for accounting interrupts

int **ccw_device_set_offline**(struct *ccw_device* * *cdev*)
disable a ccw device for I/O

Parameters

struct ccw_device * cdev target ccw device

Description

This function calls the driver's `set_offline()` function for **cdev**, if given, and then disables **cdev**.

Return

`0` on success and a negative error value on failure.

Context

enabled, ccw device lock not held

int **`ccw_device_set_online`**(struct *ccw_device* * *cdev*)
enable a ccw device for I/O

Parameters

`struct ccw_device * cdev` target ccw device

Description

This function first enables **cdev** and then calls the driver's `set_online()` function for **cdev**, if given. If `set_online()` returns an error, **cdev** is disabled again.

Return

`0` on success and a negative error value on failure.

Context

enabled, ccw device lock not held

struct *ccw_device* * **`get_ccwdev_by_dev_id`**(struct *ccw_dev_id* * *dev_id*)
obtain device from a ccw device id

Parameters

`struct ccw_dev_id * dev_id` id of the device to be searched

Description

This function searches all devices attached to the ccw bus for a device matching **dev_id**.

Return

If a device is found its reference count is increased and returned; else `NULL` is returned.

struct *ccw_device* * **`get_ccwdev_by_busid`**(struct *ccw_driver* * *cdrv*, const char * *bus_id*)
obtain device from a bus id

Parameters

`struct ccw_driver * cdrv` driver the device is owned by

`const char * bus_id` bus id of the device to be searched

Description

This function searches all devices owned by **cdrv** for a device with a bus id matching **bus_id**.

Return

If a match is found, its reference count of the found device is increased and it is returned; else `NULL` is returned.

int **`ccw_driver_register`**(struct *ccw_driver* * *cdriver*)
register a ccw driver

Parameters

`struct ccw_driver * cdriver` driver to be registered

Description

This function is mainly a wrapper around *driver_register()*.

Return

0 on success and a negative error value on failure.

void **ccw_driver_unregister**(struct *ccw_driver* * *cdriver*)
deregister a ccw driver

Parameters

struct ccw_driver * cdriver driver to be deregistered

Description

This function is mainly a wrapper around *driver_unregister()*.

int **ccw_device_siosl**(struct *ccw_device* * *cdev*)
initiate logging

Parameters

struct ccw_device * cdev ccw device

Description

This function is used to invoke model-dependent logging within the channel subsystem.

int **ccw_device_set_options_mask**(struct *ccw_device* * *cdev*, unsigned long *flags*)
set some options and unset the rest

Parameters

struct ccw_device * cdev device for which the options are to be set

unsigned long flags options to be set

Description

All flags specified in **flags** are set, all flags not specified in **flags** are cleared.

Return

0 on success, -EINVAL on an invalid flag combination.

int **ccw_device_set_options**(struct *ccw_device* * *cdev*, unsigned long *flags*)
set some options

Parameters

struct ccw_device * cdev device for which the options are to be set

unsigned long flags options to be set

Description

All flags specified in **flags** are set, the remainder is left untouched.

Return

0 on success, -EINVAL if an invalid flag combination would ensue.

void **ccw_device_clear_options**(struct *ccw_device* * *cdev*, unsigned long *flags*)
clear some options

Parameters

struct ccw_device * cdev device for which the options are to be cleared

unsigned long flags options to be cleared

Description

All flags specified in **flags** are cleared, the remainder is left untouched.

int **ccw_device_is_pathgroup**(struct *ccw_device* * *cdev*)
determine if paths to this device are grouped

Parameters

struct ccw_device * cdev ccw device

Description

Return non-zero if there is a path group, zero otherwise.

int **ccw_device_is_multipath**(struct *ccw_device* * *cdev*)
determine if device is operating in multipath mode

Parameters

struct ccw_device * cdev ccw device

Description

Return non-zero if device is operating in multipath mode, zero otherwise.

int **ccw_device_clear**(struct *ccw_device* * *cdev*, unsigned long *intparm*)
terminate I/O request processing

Parameters

struct ccw_device * cdev target ccw device

unsigned long intparm interruption parameter; value is only used if no I/O is outstanding, otherwise the intparm associated with the I/O request is returned

Description

ccw_device_clear() calls csch on **cdev**'s subchannel.

Return

`0` on success, -`ENODEV` on device not operational, -`EINVAL` on invalid device state.

Context

Interrupts disabled, ccw device lock held

int **ccw_device_start_timeout_key**(struct *ccw_device* * *cdev*, struct *ccw1* * *cpa*, unsigned long *intparm*, __u8 *lpm*, __u8 *key*, unsigned long *flags*, int *expires*)
start a s390 channel program with timeout and key

Parameters

struct ccw_device * cdev target ccw device

struct ccw1 * cpa logical start address of channel program

unsigned long intparm user specific interruption parameter; will be presented back to **cdev**'s interrupt handler. Allows a device driver to associate the interrupt with a particular I/O request.

__u8 lpm defines the channel path to be used for a specific I/O request. A value of 0 will make cio use the opm.

__u8 key storage key to be used for the I/O

unsigned long flags additional flags; defines the action to be performed for I/O processing.

int expires timeout value in jiffies

Description

Start a S/390 channel program. When the interrupt arrives, the IRQ handler is called, either immediately, delayed (dev-end missing, or sense required) or never (no IRQ handler registered). This function notifies the device driver if the channel program has not completed during the time specified by **expires**. If a timeout occurs, the channel program is terminated via xsch, hsch or csch, and the device's interrupt handler will be called with an irb containing ERR_PTR(-ETIMEDOUT).

Return

0, if the operation was successful; -EBUSY, if the device is busy, or status pending; -EACCES, if no path specified in **lpm** is operational; -ENODEV, if the device is not operational.

Context

Interrupts disabled, ccw device lock held

int **ccw_device_start_key**(struct *ccw_device* * *cdev*, struct *ccw1* * *cpa*, unsigned long *intparm*, __u8 *lpm*, __u8 *key*, unsigned long *flags*)
start a s390 channel program with key

Parameters

struct ccw_device * cdev target ccw device

struct ccw1 * cpa logical start address of channel program

unsigned long intparm user specific interruption parameter; will be presented back to **cdev**'s interrupt handler. Allows a device driver to associate the interrupt with a particular I/O request.

__u8 lpm defines the channel path to be used for a specific I/O request. A value of 0 will make cio use the opm.

__u8 key storage key to be used for the I/O

unsigned long flags additional flags; defines the action to be performed for I/O processing.

Description

Start a S/390 channel program. When the interrupt arrives, the IRQ handler is called, either immediately, delayed (dev-end missing, or sense required) or never (no IRQ handler registered).

Return

0, if the operation was successful; -EBUSY, if the device is busy, or status pending; -EACCES, if no path specified in **lpm** is operational; -ENODEV, if the device is not operational.

Context

Interrupts disabled, ccw device lock held

int **ccw_device_start**(struct *ccw_device* * *cdev*, struct *ccw1* * *cpa*, unsigned long *intparm*, __u8 *lpm*, unsigned long *flags*)
start a s390 channel program

Parameters

struct ccw_device * cdev target ccw device

struct ccw1 * cpa logical start address of channel program

unsigned long intparm user specific interruption parameter; will be presented back to **cdev**'s interrupt handler. Allows a device driver to associate the interrupt with a particular I/O request.

__u8 lpm defines the channel path to be used for a specific I/O request. A value of 0 will make cio use the opm.

unsigned long flags additional flags; defines the action to be performed for I/O processing.

Description

Start a S/390 channel program. When the interrupt arrives, the IRQ handler is called, either immediately, delayed (dev-end missing, or sense required) or never (no IRQ handler registered).

Return

0, if the operation was successful; -EBUSY, if the device is busy, or status pending; -EACCES, if no path specified in **lpm** is operational; -ENODEV, if the device is not operational.

Context

Interrupts disabled, ccw device lock held

int **ccw_device_start_timeout**(struct *ccw_device* * *cdev*, struct *ccw1* * *cpa*, unsigned long *intparm*, __u8 *lpm*, unsigned long *flags*, int *expires*)
start a s390 channel program with timeout

Parameters

struct ccw_device * cdev target ccw device

struct ccw1 * cpa logical start address of channel program

unsigned long intparm user specific interruption parameter; will be presented back to **cdev**'s interrupt handler. Allows a device driver to associate the interrupt with a particular I/O request.

__u8 lpm defines the channel path to be used for a specific I/O request. A value of 0 will make cio use the opm.

unsigned long flags additional flags; defines the action to be performed for I/O processing.

int expires timeout value in jiffies

Description

Start a S/390 channel program. When the interrupt arrives, the IRQ handler is called, either immediately, delayed (dev-end missing, or sense required) or never (no IRQ handler registered). This function notifies the device driver if the channel program has not completed during the time specified by **expires**. If a timeout occurs, the channel program is terminated via xsch, hsch or csch, and the device's interrupt handler will be called with an irb containing ERR_PTR(-ETIMEDOUT).

Return

0, if the operation was successful; -EBUSY, if the device is busy, or status pending; -EACCES, if no path specified in **lpm** is operational; -ENODEV, if the device is not operational.

Context

Interrupts disabled, ccw device lock held

int **ccw_device_halt**(struct *ccw_device* * *cdev*, unsigned long *intparm*)
halt I/O request processing

Parameters

struct ccw_device * cdev target ccw device

unsigned long intparm interruption parameter; value is only used if no I/O is outstanding, otherwise the intparm associated with the I/O request is returned

Description

ccw_device_halt() calls hsch on **cdev**'s subchannel.

Return

0 on success, -ENODEV on device not operational, -EINVAL on invalid device state, -EBUSY on device busy or interrupt pending.

Context

Interrupts disabled, ccw device lock held

int **ccw_device_resume**(struct *ccw_device* * *cdev*)
resume channel program execution

Parameters

struct ccw_device * cdev target ccw device

Description

`ccw_device_resume()` calls rsch on **cdev**'s subchannel.

Return

0 on success, -ENODEV on device not operational, -EINVAL on invalid device state, -EBUSY on device busy or interrupt pending.

Context

Interrupts disabled, ccw device lock held

struct *ciw* * **ccw_device_get_ciw**(struct *ccw_device* * *cdev*, __u32 *ct*)
Search for CIW command in extended sense data.

Parameters

struct ccw_device * cdev ccw device to inspect

__u32 ct command type to look for

Description

During SenseID, command information words (CIWs) describing special commands available to the device may have been stored in the extended sense data. This function searches for CIWs of a specified command type in the extended sense data.

Return

NULL if no extended sense data has been stored or if no CIW of the specified command type could be found, else a pointer to the CIW of the specified command type.

__u8 **ccw_device_get_path_mask**(struct *ccw_device* * *cdev*)
get currently available paths

Parameters

struct ccw_device * cdev ccw device to be queried

Return

0 if no subchannel for the device is available, else the mask of currently available paths for the ccw device's subchannel.

struct channel_path_desc * **ccw_device_get_chp_desc**(struct *ccw_device* * *cdev*, int *chp_idx*)
return newly allocated channel-path descriptor

Parameters

struct ccw_device * cdev device to obtain the descriptor for

int chp_idx index of the channel path

Description

On success return a newly allocated copy of the channel-path description data associated with the given channel path. Return NULL on error.

void **ccw_device_get_id**(struct *ccw_device* * *cdev*, struct *ccw_dev_id* * *dev_id*)
obtain a ccw device id

Parameters

struct ccw_device * cdev device to obtain the id for

struct ccw_dev_id * dev_id where to fill in the values

int **ccw_device_tm_start_timeout_key**(struct *ccw_device* * *cdev*, struct tcw * *tcw*, unsigned long *intparm*, u8 *lpm*, u8 *key*, int *expires*)
perform start function

Parameters

struct ccw_device * cdev ccw device on which to perform the start function

struct tcw * tcw transport-command word to be started

unsigned long intparm user defined parameter to be passed to the interrupt handler

u8 lpm mask of paths to use

u8 key storage key to use for storage access

int expires time span in jiffies after which to abort request

Description

Start the tcw on the given ccw device. Return zero on success, non-zero otherwise.

int **ccw_device_tm_start_key**(struct *ccw_device* * *cdev*, struct tcw * *tcw*, unsigned long *intparm*, u8 *lpm*, u8 *key*)
perform start function

Parameters

struct ccw_device * cdev ccw device on which to perform the start function

struct tcw * tcw transport-command word to be started

unsigned long intparm user defined parameter to be passed to the interrupt handler

u8 lpm mask of paths to use

u8 key storage key to use for storage access

Description

Start the tcw on the given ccw device. Return zero on success, non-zero otherwise.

int **ccw_device_tm_start**(struct *ccw_device* * *cdev*, struct tcw * *tcw*, unsigned long *intparm*, u8 *lpm*)
perform start function

Parameters

struct ccw_device * cdev ccw device on which to perform the start function

struct tcw * tcw transport-command word to be started

unsigned long intparm user defined parameter to be passed to the interrupt handler

u8 lpm mask of paths to use

Description

Start the tcw on the given ccw device. Return zero on success, non-zero otherwise.

int **ccw_device_tm_start_timeout**(struct *ccw_device* * *cdev*, struct tcw * *tcw*, unsigned long *intparm*, u8 *lpm*, int *expires*)
perform start function

Parameters

struct ccw_device * cdev ccw device on which to perform the start function

struct tcw * tcw transport-command word to be started

unsigned long intparm user defined parameter to be passed to the interrupt handler

u8 lpm mask of paths to use

int expires time span in jiffies after which to abort request

Description

Start the tcw on the given ccw device. Return zero on success, non-zero otherwise.

int **ccw_device_get_mdc**(struct *ccw_device* * *cdev*, u8 *mask*)
accumulate max data count

Parameters

struct ccw_device * cdev ccw device for which the max data count is accumulated

u8 mask mask of paths to use

Description

Return the number of 64K-bytes blocks all paths at least support for a transport command. Return values <= 0 indicate failures.

int **ccw_device_tm_intrg**(struct *ccw_device* * *cdev*)
perform interrogate function

Parameters

struct ccw_device * cdev ccw device on which to perform the interrogate function

Description

Perform an interrogate function on the given ccw device. Return zero on success, non-zero otherwise.

void **ccw_device_get_schid**(struct *ccw_device* * *cdev*, struct subchannel_id * *schid*)
obtain a subchannel id

Parameters

struct ccw_device * cdev device to obtain the id for

struct subchannel_id * schid where to fill in the values

The channel-measurement facility

The channel-measurement facility provides a means to collect measurement data which is made available by the channel subsystem for each channel attached device.

struct **cmbdata**
channel measurement block data for user space

Definition

```
struct cmbdata {
  __u64 size;
  __u64 elapsed_time;
  __u64 ssch_rsch_count;
  __u64 sample_count;
  __u64 device_connect_time;
  __u64 function_pending_time;
  __u64 device_disconnect_time;
  __u64 control_unit_queuing_time;
  __u64 device_active_only_time;
  __u64 device_busy_time;
  __u64 initial_command_response_time;
};
```

Members

size size of the stored data

elapsed_time time since last sampling

ssch_rsch_count number of ssch and rsch

sample_count number of samples

device_connect_time time of device connect

`function_pending_time` time of function pending

`device_disconnect_time` time of device disconnect

`control_unit_queuing_time` time of control unit queuing

`device_active_only_time` time of device active only

`device_busy_time` time of device busy (ext. format)

`initial_command_response_time` initial command response time (ext. format)

Description

All values are stored as 64 bit for simplicity, especially in 32 bit emulation mode. All time values are normalized to nanoseconds. Currently, two formats are known, which differ by the size of this structure, i.e. the last two members are only set when the extended channel measurement facility (first shipped in z990 machines) is activated. Potentially, more fields could be added, which would result in a new ioctl number.

int **`enable_cmf`**(struct *ccw_device* * *cdev*)
    switch on the channel measurement for a specific device

Parameters

`struct ccw_device * cdev` The ccw device to be enabled

Description

> Enable channel measurements for **cdev**. If this is called on a device for which channel measurement is already enabled a reset of the measurement data is triggered.

Return

`0` for success or a negative error value.

Context

non-atomic

int **`disable_cmf`**(struct *ccw_device* * *cdev*)
    switch off the channel measurement for a specific device

Parameters

`struct ccw_device * cdev` The ccw device to be disabled

Return

`0` for success or a negative error value.

Context

non-atomic

u64 **`cmf_read`**(struct *ccw_device* * *cdev*, int *index*)
    read one value from the current channel measurement block

Parameters

`struct ccw_device * cdev` the channel to be read

`int index` the index of the value to be read

Return

The value read or `0` if the value cannot be read.

Context

any

int **`cmf_readall`**(struct *ccw_device* * *cdev*, struct *cmbdata* * *data*)
    read the current channel measurement block

Parameters

struct ccw_device * cdev the channel to be read

struct cmbdata * data a pointer to a data block that will be filled

Return

0 on success, a negative error value otherwise.

Context

any

The ccwgroup bus

The ccwgroup bus only contains artificial devices, created by the user. Many networking devices (e.g. qeth) are in fact composed of several ccw devices (like read, write and data channel for qeth). The ccwgroup bus provides a mechanism to create a meta-device which contains those ccw devices as slave devices and can be associated with the netdevice.

ccw group devices

struct **ccwgroup_device**
    ccw group device

Definition

```
struct ccwgroup_device {
  enum {
    CCWGROUP_OFFLINE,
    CCWGROUP_ONLINE,
  } state;
  unsigned int count;
  struct device   dev;
  struct work_struct ungroup_work;
  struct ccw_device *cdev[0];
};
```

Members

state online/offline state

count number of attached slave devices

dev embedded device structure

ungroup_work work to be done when a ccwgroup notifier has action type BUS_NOTIFY_UNBIND_DRIVER

cdev variable number of slave devices, allocated as needed

struct **ccwgroup_driver**
    driver for ccw group devices

Definition

```
struct ccwgroup_driver {
  int (*setup) (struct ccwgroup_device *);
  void (*remove) (struct ccwgroup_device *);
  int (*set_online) (struct ccwgroup_device *);
  int (*set_offline) (struct ccwgroup_device *);
  void (*shutdown)(struct ccwgroup_device *);
  int (*prepare) (struct ccwgroup_device *);
  void (*complete) (struct ccwgroup_device *);
  int (*freeze)(struct ccwgroup_device *);
  int (*thaw) (struct ccwgroup_device *);
  int (*restore)(struct ccwgroup_device *);
  struct device_driver driver;
  struct ccw_driver *ccw_driver;
};
```

Members

setup function called during device creation to setup the device

remove function called on remove

set_online function called when device is set online

set_offline function called when device is set offline

shutdown function called when device is shut down

prepare prepare for pm state transition

complete undo work done in **prepare**

freeze callback for freezing during hibernation snapshotting

thaw undo work done in **freeze**

restore callback for restoring after hibernation

driver embedded driver structure

ccw_driver supported ccw_driver (optional)

int **ccwgroup_set_online**(struct *ccwgroup_device* * *gdev*)
enable a ccwgroup device

Parameters

struct ccwgroup_device * gdev target ccwgroup device

Description

This function attempts to put the ccwgroup device into the online state.

Return

0 on success and a negative error value on failure.

int **ccwgroup_set_offline**(struct *ccwgroup_device* * *gdev*)
disable a ccwgroup device

Parameters

struct ccwgroup_device * gdev target ccwgroup device

Description

This function attempts to put the ccwgroup device into the offline state.

Return

0 on success and a negative error value on failure.

int **ccwgroup_create_dev**(struct *device* * *parent*, struct *ccwgroup_driver* * *gdrv*, int *num_devices*, const char * *buf*)
create and register a ccw group device

Parameters

struct device * parent parent device for the new device

struct ccwgroup_driver * gdrv driver for the new group device

int num_devices number of slave devices

const char * buf buffer containing comma separated bus ids of slave devices

Description

Create and register a new ccw group device as a child of **parent**. Slave devices are obtained from the list of bus ids given in **buf**.

Return

`0` on success and an error code on failure.

Context

non-atomic

int **ccwgroup_driver_register**(struct *ccwgroup_driver* * *cdriver*)
register a ccw group driver

Parameters

struct ccwgroup_driver * cdriver driver to be registered

Description

This function is mainly a wrapper around *driver_register()*.

void **ccwgroup_driver_unregister**(struct *ccwgroup_driver* * *cdriver*)
deregister a ccw group driver

Parameters

struct ccwgroup_driver * cdriver driver to be deregistered

Description

This function is mainly a wrapper around *driver_unregister()*.

int **ccwgroup_probe_ccwdev**(struct *ccw_device* * *cdev*)
probe function for slave devices

Parameters

struct ccw_device * cdev ccw device to be probed

Description

This is a dummy probe function for ccw devices that are slave devices in a ccw group device.

Return

always `0`

void **ccwgroup_remove_ccwdev**(struct *ccw_device* * *cdev*)
remove function for slave devices

Parameters

struct ccw_device * cdev ccw device to be removed

Description

This is a remove function for ccw devices that are slave devices in a ccw group device. It sets the ccw device offline and also deregisters the embedding ccw group device.

Generic interfaces

The following section contains interfaces in use not only by drivers dealing with ccw devices, but drivers for various other s390 hardware as well.

Adapter interrupts

The common I/O layer provides helper functions for dealing with adapter interrupts and interrupt vectors.

int **register_adapter_interrupt**(struct airq_struct * *airq*)
register adapter interrupt handler

Parameters

struct airq_struct * airq pointer to adapter interrupt descriptor

Description

Returns 0 on success, or -EINVAL.

void **unregister_adapter_interrupt**(struct airq_struct * *airq*)
unregister adapter interrupt handler

Parameters

struct airq_struct * airq pointer to adapter interrupt descriptor

struct airq_iv * **airq_iv_create**(unsigned long *bits*, unsigned long *flags*)
create an interrupt vector

Parameters

unsigned long bits number of bits in the interrupt vector

unsigned long flags allocation flags

Description

Returns a pointer to an interrupt vector structure

void **airq_iv_release**(struct airq_iv * *iv*)
release an interrupt vector

Parameters

struct airq_iv * iv pointer to interrupt vector structure

unsigned long **airq_iv_alloc**(struct airq_iv * *iv*, unsigned long *num*)
allocate irq bits from an interrupt vector

Parameters

struct airq_iv * iv pointer to an interrupt vector structure

unsigned long num number of consecutive irq bits to allocate

Description

Returns the bit number of the first irq in the allocated block of irqs, or -1UL if no bit is available or the AIRQ_IV_ALLOC flag has not been specified

void **airq_iv_free**(struct airq_iv * *iv*, unsigned long *bit*, unsigned long *num*)
free irq bits of an interrupt vector

Parameters

struct airq_iv * iv pointer to interrupt vector structure

unsigned long bit number of the first irq bit to free

unsigned long num number of consecutive irq bits to free

unsigned long **airq_iv_scan**(struct airq_iv * *iv*, unsigned long *start*, unsigned long *end*)
scan interrupt vector for non-zero bits

Parameters

struct airq_iv * iv pointer to interrupt vector structure

unsigned long start bit number to start the search

unsigned long end bit number to end the search

Description

Returns the bit number of the next non-zero interrupt bit, or -1UL if the scan completed without finding any more any non-zero bits.

CHAPTER

TWENTYNINE

VME DEVICE DRIVERS

Driver registration

As with other subsystems within the Linux kernel, VME device drivers register with the VME subsystem, typically called from the devices init routine. This is achieved via a call to *vme_register_driver()*.

A pointer to a structure of type *struct vme_driver* must be provided to the registration function. Along with the maximum number of devices your driver is able to support.

At the minimum, the '.name', '.match' and '.probe' elements of *struct vme_driver* should be correctly set. The '.name' element is a pointer to a string holding the device driver's name.

The '.match' function allows control over which VME devices should be registered with the driver. The match function should return 1 if a device should be probed and 0 otherwise. This example match function (from vme_user.c) limits the number of devices probed to one:

```
#define USER_BUS_MAX    1
...
static int vme_user_match(struct vme_dev *vdev)
{
        if (vdev->id.num >= USER_BUS_MAX)
                return 0;
        return 1;
}
```

The '.probe' element should contain a pointer to the probe routine. The probe routine is passed a *struct vme_dev* pointer as an argument.

Here, the 'num' field refers to the sequential device ID for this specific driver. The bridge number (or bus number) can be accessed using dev->bridge->num.

A function is also provided to unregister the driver from the VME core called *vme_unregister_driver()* and should usually be called from the device driver's exit routine.

Resource management

Once a driver has registered with the VME core the provided match routine will be called the number of times specified during the registration. If a match succeeds, a non-zero value should be returned. A zero return value indicates failure. For all successful matches, the probe routine of the corresponding driver is called. The probe routine is passed a pointer to the devices device structure. This pointer should be saved, it will be required for requesting VME resources.

The driver can request ownership of one or more master windows (*vme_master_request()*), slave windows (*vme_slave_request()*) and/or dma channels (*vme_dma_request()*). Rather than allowing the device driver to request a specific window or DMA channel (which may be used by a different driver) the API allows a resource to be assigned based on the required attributes of the driver in question. For slave windows these attributes are split into the VME address spaces that need to be accessed in 'aspace' and VME bus cycle types required in 'cycle'. Master windows add a further set of attributes in 'width' specifying the required data transfer widths. These attributes are defined as bitmasks and as such any combination of the attributes can be requested for a single window, the core will assign a window that meets the requirements, returning a pointer of type vme_resource that should be used to identify the allocated resource when it is used. For DMA controllers, the request function requires the potential direction of any transfers to be provided in the route attributes. This is typically VME-to-MEM and/or MEM-to-VME, though some hardware can support VME-to-VME and MEM-to-MEM transfers as well as test pattern generation. If an unallocated window fitting the requirements can not be found a NULL pointer will be returned.

Functions are also provided to free window allocations once they are no longer required. These functions (*vme_master_free()*, *vme_slave_free()* and *vme_dma_free()*) should be passed the pointer to the resource provided during resource allocation.

Master windows

Master windows provide access from the local processor[s] out onto the VME bus. The number of windows available and the available access modes is dependent on the underlying chipset. A window must be configured before it can be used.

Master window configuration

Once a master window has been assigned *vme_master_set()* can be used to configure it and *vme_master_get()* to retrieve the current settings. The address spaces, transfer widths and cycle types are the same as described under resource management, however some of the options are mutually exclusive. For example, only one address space may be specified.

Master window access

The function *vme_master_read()* can be used to read from and *vme_master_write()* used to write to configured master windows.

In addition to simple reads and writes, *vme_master_rmw()* is provided to do a read-modify-write transaction. Parts of a VME window can also be mapped into user space memory using *vme_master_mmap()*.

Slave windows

Slave windows provide devices on the VME bus access into mapped portions of the local memory. The number of windows available and the access modes that can be used is dependent on the underlying chipset. A window must be configured before it can be used.

Slave window configuration

Once a slave window has been assigned *vme_slave_set()* can be used to configure it and *vme_slave_get()* to retrieve the current settings.

The address spaces, transfer widths and cycle types are the same as described under resource management, however some of the options are mutually exclusive. For example, only one address space may be specified.

Slave window buffer allocation

Functions are provided to allow the user to allocate (*vme_alloc_consistent()*) and free (*vme_free_consistent()*) contiguous buffers which will be accessible by the VME bridge. These functions do not have to be used, other methods can be used to allocate a buffer, though care must be taken to ensure that they are contiguous and accessible by the VME bridge.

Slave window access

Slave windows map local memory onto the VME bus, the standard methods for accessing memory should be used.

DMA channels

The VME DMA transfer provides the ability to run link-list DMA transfers. The API introduces the concept of DMA lists. Each DMA list is a link-list which can be passed to a DMA controller. Multiple lists can be created, extended, executed, reused and destroyed.

List Management

The function *vme_new_dma_list()* is provided to create and *vme_dma_list_free()* to destroy DMA lists. Execution of a list will not automatically destroy the list, thus enabling a list to be reused for repetitive tasks.

List Population

An item can be added to a list using *vme_dma_list_add()* (the source and destination attributes need to be created before calling this function, this is covered under “Transfer Attributes”).

Note:

The detailed attributes of the transfers source and destination are not checked until an entry is added to a DMA list, the request for a DMA channel purely checks the directions in which the controller is expected to transfer data. As a result it is possible for this call to return an error, for example if the source or destination is in an unsupported VME address space.

Transfer Attributes

The attributes for the source and destination are handled separately from adding an item to a list. This is due to the diverse attributes required for each type of source and destination. There are functions to create attributes for PCI, VME and pattern sources and destinations (where appropriate):

- PCI source or destination: *vme_dma_pci_attribute()*
- VME source or destination: *vme_dma_vme_attribute()*
- Pattern source: *vme_dma_pattern_attribute()*

The function *vme_dma_free_attribute()* should be used to free an attribute.

List Execution

The function *vme_dma_list_exec()* queues a list for execution and will return once the list has been executed.

Interrupts

The VME API provides functions to attach and detach callbacks to specific VME level and status ID combinations and for the generation of VME interrupts with specific VME level and status IDs.

Attaching Interrupt Handlers

The function *vme_irq_request()* can be used to attach and *vme_irq_free()* to free a specific VME level and status ID combination. Any given combination can only be assigned a single callback function. A void pointer parameter is provided, the value of which is passed to the callback function, the use of this pointer is user undefined. The callback parameters are as follows. Care must be taken in writing a callback function, callback functions run in interrupt context:

```
void callback(int level, int statid, void *priv);
```

Interrupt Generation

The function *vme_irq_generate()* can be used to generate a VME interrupt at a given VME level and VME status ID.

Location monitors

The VME API provides the following functionality to configure the location monitor.

Location Monitor Management

The function *vme_lm_request()* is provided to request the use of a block of location monitors and *vme_lm_free()* to free them after they are no longer required. Each block may provide a number of location monitors, monitoring adjacent locations. The function *vme_lm_count()* can be used to determine how many locations are provided.

Location Monitor Configuration

Once a bank of location monitors has been allocated, the function *vme_lm_set()* is provided to configure the location and mode of the location monitor. The function *vme_lm_get()* can be used to retrieve existing settings.

Location Monitor Use

The function *vme_lm_attach()* enables a callback to be attached and *vme_lm_detach()* allows on to be detached from each location monitor location. Each location monitor can monitor a number of adjacent locations. The callback function is declared as follows.

```
void callback(void *data);
```

Slot Detection

The function *vme_slot_num()* returns the slot ID of the provided bridge.

Bus Detection

The function *vme_bus_num()* returns the bus ID of the provided bridge.

VME API

struct **vme_dev**
Structure representing a VME device

Definition

```
struct vme_dev {
  int num;
  struct vme_bridge *bridge;
  struct device dev;
  struct list_head drv_list;
  struct list_head bridge_list;
};
```

Members

num The device number

bridge Pointer to the bridge device this device is on

dev Internal device structure

drv_list List of devices (per driver)

bridge_list List of devices (per bridge)

struct **vme_driver**
Structure representing a VME driver

Definition

```
struct vme_driver {
  const char *name;
  int (*match)(struct vme_dev *);
  int (*probe)(struct vme_dev *);
  int (*remove)(struct vme_dev *);
  struct device_driver driver;
  struct list_head devices;
};
```

Members

name Driver name, should be unique among VME drivers and usually the same as the module name.

match Callback used to determine whether probe should be run.

probe Callback for device binding, called when new device is detected.

remove Callback, called on device removal.

driver Underlying generic device driver structure.

devices List of VME devices (struct vme_dev) associated with this driver.

void * **vme_alloc_consistent**(struct vme_resource * *resource*, size_t *size*, dma_addr_t * *dma*)
Allocate contiguous memory.

Parameters

struct vme_resource * resource Pointer to VME resource.

size_t size Size of allocation required.

dma_addr_t * dma Pointer to variable to store physical address of allocation.

Description

Allocate a contiguous block of memory for use by the driver. This is used to create the buffers for the slave windows.

Return

Virtual address of allocation on success, NULL on failure.

void **vme_free_consistent**(struct vme_resource * *resource*, size_t *size*, void * *vaddr*, dma_addr_t *dma*)
Free previously allocated memory.

Parameters

struct vme_resource * resource Pointer to VME resource.

size_t size Size of allocation to free.

void * vaddr Virtual address of allocation.

dma_addr_t dma Physical address of allocation.

Description

Free previously allocated block of contiguous memory.

size_t **vme_get_size**(struct vme_resource * *resource*)
Helper function returning size of a VME window

Parameters

struct vme_resource * resource Pointer to VME slave or master resource.

Description

Determine the size of the VME window provided. This is a helper function, wrappering the call to vme_master_get or vme_slave_get depending on the type of window resource handed to it.

Return

Size of the window on success, zero on failure.

struct vme_resource * **vme_slave_request**(struct *vme_dev* * *vdev*, u32 *address*, u32 *cycle*)
Request a VME slave window resource.

Parameters

struct vme_dev * vdev Pointer to VME device struct vme_dev assigned to driver instance.

u32 address Required VME address space.

u32 cycle Required VME data transfer cycle type.

Description

Request use of a VME window resource capable of being set for the requested address space and data transfer cycle.

Return

Pointer to VME resource on success, NULL on failure.

int **vme_slave_set**(struct vme_resource * *resource*, int *enabled*, unsigned long long *vme_base*, unsigned long long *size*, dma_addr_t *buf_base*, u32 *aspace*, u32 *cycle*)
Set VME slave window configuration.

Parameters

struct vme_resource * resource Pointer to VME slave resource.

int enabled State to which the window should be configured.

`unsigned long long vme_base` Base address for the window.

`unsigned long long size` Size of the VME window.

`dma_addr_t buf_base` Based address of buffer used to provide VME slave window storage.

`u32 aspace` VME address space for the VME window.

`u32 cycle` VME data transfer cycle type for the VME window.

Description

Set configuration for provided VME slave window.

Return

Zero on success, -EINVAL if operation is not supported on this device, if an invalid resource has been provided or invalid attributes are provided. Hardware specific errors may also be returned.

int **`vme_slave_get`**(struct vme_resource * *resource*, int * *enabled*, unsigned long long * *vme_base*, unsigned long long * *size*, dma_addr_t * *buf_base*, u32 * *aspace*, u32 * *cycle*)
Retrieve VME slave window configuration.

Parameters

`struct vme_resource * resource` Pointer to VME slave resource.

`int * enabled` Pointer to variable for storing state.

`unsigned long long * vme_base` Pointer to variable for storing window base address.

`unsigned long long * size` Pointer to variable for storing window size.

`dma_addr_t * buf_base` Pointer to variable for storing slave buffer base address.

`u32 * aspace` Pointer to variable for storing VME address space.

`u32 * cycle` Pointer to variable for storing VME data transfer cycle type.

Description

Return configuration for provided VME slave window.

Return

Zero on success, -EINVAL if operation is not supported on this device or if an invalid resource has been provided.

void **`vme_slave_free`**(struct vme_resource * *resource*)
Free VME slave window

Parameters

`struct vme_resource * resource` Pointer to VME slave resource.

Description

Free the provided slave resource so that it may be reallocated.

struct vme_resource * **`vme_master_request`**(struct *vme_dev* * *vdev*, u32 *address*, u32 *cycle*, u32 *dwidth*)
Request a VME master window resource.

Parameters

`struct vme_dev * vdev` Pointer to VME device struct vme_dev assigned to driver instance.

`u32 address` Required VME address space.

`u32 cycle` Required VME data transfer cycle type.

`u32 dwidth` Required VME data transfer width.

Description

Request use of a VME window resource capable of being set for the requested address space, data transfer cycle and width.

Return

Pointer to VME resource on success, NULL on failure.

int **vme_master_set**(struct vme_resource * *resource*, int *enabled*, unsigned long long *vme_base*, unsigned long long *size*, u32 *aspace*, u32 *cycle*, u32 *dwidth*)
Set VME master window configuration.

Parameters

struct vme_resource * resource Pointer to VME master resource.

int enabled State to which the window should be configured.

unsigned long long vme_base Base address for the window.

unsigned long long size Size of the VME window.

u32 aspace VME address space for the VME window.

u32 cycle VME data transfer cycle type for the VME window.

u32 dwidth VME data transfer width for the VME window.

Description

Set configuration for provided VME master window.

Return

Zero on success, -EINVAL if operation is not supported on this device, if an invalid resource has been provided or invalid attributes are provided. Hardware specific errors may also be returned.

int **vme_master_get**(struct vme_resource * *resource*, int * *enabled*, unsigned long long * *vme_base*, unsigned long long * *size*, u32 * *aspace*, u32 * *cycle*, u32 * *dwidth*)
Retrieve VME master window configuration.

Parameters

struct vme_resource * resource Pointer to VME master resource.

int * enabled Pointer to variable for storing state.

unsigned long long * vme_base Pointer to variable for storing window base address.

unsigned long long * size Pointer to variable for storing window size.

u32 * aspace Pointer to variable for storing VME address space.

u32 * cycle Pointer to variable for storing VME data transfer cycle type.

u32 * dwidth Pointer to variable for storing VME data transfer width.

Description

Return configuration for provided VME master window.

Return

Zero on success, -EINVAL if operation is not supported on this device or if an invalid resource has been provided.

ssize_t **vme_master_read**(struct vme_resource * *resource*, void * *buf*, size_t *count*, loff_t *offset*)
Read data from VME space into a buffer.

Parameters

struct vme_resource * resource Pointer to VME master resource.

void * buf Pointer to buffer where data should be transferred.

size_t count Number of bytes to transfer.

loff_t offset Offset into VME master window at which to start transfer.

Description

Perform read of count bytes of data from location on VME bus which maps into the VME master window at offset to buf.

Return

Number of bytes read, -EINVAL if resource is not a VME master resource or read operation is not supported. -EFAULT returned if invalid offset is provided. Hardware specific errors may also be returned.

ssize_t **vme_master_write**(struct vme_resource * *resource*, void * *buf*, size_t *count*, loff_t *offset*)
Write data out to VME space from a buffer.

Parameters

struct vme_resource * resource Pointer to VME master resource.

void * buf Pointer to buffer holding data to transfer.

size_t count Number of bytes to transfer.

loff_t offset Offset into VME master window at which to start transfer.

Description

Perform write of count bytes of data from buf to location on VME bus which maps into the VME master window at offset.

Return

Number of bytes written, -EINVAL if resource is not a VME master resource or write operation is not supported. -EFAULT returned if invalid offset is provided. Hardware specific errors may also be returned.

unsigned int **vme_master_rmw**(struct vme_resource * *resource*, unsigned int *mask*, unsigned int *compare*, unsigned int *swap*, loff_t *offset*)
Perform read-modify-write cycle.

Parameters

struct vme_resource * resource Pointer to VME master resource.

unsigned int mask Bits to be compared and swapped in operation.

unsigned int compare Bits to be compared with data read from offset.

unsigned int swap Bits to be swapped in data read from offset.

loff_t offset Offset into VME master window at which to perform operation.

Description

Perform read-modify-write cycle on provided location: - Location on VME bus is read. - Bits selected by mask are compared with compare. - Where a selected bit matches that in compare and are selected in swap, the bit is swapped. - Result written back to location on VME bus.

Return

Bytes written on success, -EINVAL if resource is not a VME master resource or RMW operation is not supported. Hardware specific errors may also be returned.

int **vme_master_mmap**(struct vme_resource * *resource*, struct vm_area_struct * *vma*)
Mmap region of VME master window.

Parameters

struct vme_resource * resource Pointer to VME master resource.

struct vm_area_struct * vma Pointer to definition of user mapping.

Description

Memory map a region of the VME master window into user space.

Return

Zero on success, -EINVAL if resource is not a VME master resource or -EFAULT if map exceeds window size. Other generic mmap errors may also be returned.

void **vme_master_free**(struct vme_resource * *resource*)
Free VME master window

Parameters

struct vme_resource * resource Pointer to VME master resource.

Description

Free the provided master resource so that it may be reallocated.

struct vme_resource * **vme_dma_request**(struct *vme_dev* * *vdev*, u32 *route*)
Request a DMA controller.

Parameters

struct vme_dev * vdev Pointer to VME device struct vme_dev assigned to driver instance.

u32 route Required src/destination combination.

Description

Request a VME DMA controller with capability to perform transfers bewteen requested source/destination combination.

Return

Pointer to VME DMA resource on success, NULL on failure.

struct vme_dma_list * **vme_new_dma_list**(struct vme_resource * *resource*)
Create new VME DMA list.

Parameters

struct vme_resource * resource Pointer to VME DMA resource.

Description

Create a new VME DMA list. It is the responsibility of the user to free the list once it is no longer required with *vme_dma_list_free()*.

Return

Pointer to new VME DMA list, NULL on allocation failure or invalid VME DMA resource.

struct vme_dma_attr * **vme_dma_pattern_attribute**(u32 *pattern*, u32 *type*)
Create "Pattern" type VME DMA list attribute.

Parameters

u32 pattern Value to use used as pattern

u32 type Type of pattern to be written.

Description

Create VME DMA list attribute for pattern generation. It is the responsibility of the user to free used attributes using *vme_dma_free_attribute()*.

Return

Pointer to VME DMA attribute, NULL on failure.

struct vme_dma_attr * **vme_dma_pci_attribute**(dma_addr_t *address*)
Create "PCI" type VME DMA list attribute.

Parameters

dma_addr_t address PCI base address for DMA transfer.

Description

Create VME DMA list attribute pointing to a location on PCI for DMA transfers. It is the responsibility of the user to free used attributes using *vme_dma_free_attribute()*.

Return

Pointer to VME DMA attribute, NULL on failure.

struct vme_dma_attr * **vme_dma_vme_attribute**(unsigned long long *address*, u32 *aspace*, u32 *cycle*, u32 *dwidth*)
Create "VME" type VME DMA list attribute.

Parameters

unsigned long long address VME base address for DMA transfer.

u32 aspace VME address space to use for DMA transfer.

u32 cycle VME bus cycle to use for DMA transfer.

u32 dwidth VME data width to use for DMA transfer.

Description

Create VME DMA list attribute pointing to a location on the VME bus for DMA transfers. It is the responsibility of the user to free used attributes using *vme_dma_free_attribute()*.

Return

Pointer to VME DMA attribute, NULL on failure.

void **vme_dma_free_attribute**(struct vme_dma_attr * *attributes*)
Free DMA list attribute.

Parameters

struct vme_dma_attr * attributes Pointer to DMA list attribute.

Description

Free VME DMA list attribute. VME DMA list attributes can be safely freed once *vme_dma_list_add()* has returned.

int **vme_dma_list_add**(struct vme_dma_list * *list*, struct vme_dma_attr * *src*, struct vme_dma_attr * *dest*, size_t *count*)
Add enty to a VME DMA list.

Parameters

struct vme_dma_list * list Pointer to VME list.

struct vme_dma_attr * src Pointer to DMA list attribute to use as source.

struct vme_dma_attr * dest Pointer to DMA list attribute to use as destination.

size_t count Number of bytes to transfer.

Description

Add an entry to the provided VME DMA list. Entry requires pointers to source and destination DMA attributes and a count.

Please note, the attributes supported as source and destinations for transfers are hardware dependent.

Return

Zero on success, -EINVAL if operation is not supported on this device or if the link list has already been submitted for execution. Hardware specific errors also possible.

int **vme_dma_list_exec**(struct vme_dma_list * *list*)
Queue a VME DMA list for execution.

Parameters

struct vme_dma_list * list Pointer to VME list.

Description

Queue the provided VME DMA list for execution. The call will return once the list has been executed.

Return

Zero on success, -EINVAL if operation is not supported on this device. Hardware specific errors also possible.

int **vme_dma_list_free**(struct vme_dma_list * *list*)
Free a VME DMA list.

Parameters

struct vme_dma_list * list Pointer to VME list.

Description

Free the provided DMA list and all its entries.

Return

Zero on success, -EINVAL on invalid VME resource, -EBUSY if resource is still in use. Hardware specific errors also possible.

int **vme_dma_free**(struct vme_resource * *resource*)
Free a VME DMA resource.

Parameters

struct vme_resource * resource Pointer to VME DMA resource.

Description

Free the provided DMA resource so that it may be reallocated.

Return

Zero on success, -EINVAL on invalid VME resource, -EBUSY if resource is still active.

int **vme_irq_request**(struct *vme_dev* * *vdev*, int *level*, int *statid*, void (*callback) *(int*, int, void *, void * *priv_data*)
Request a specific VME interrupt.

Parameters

struct vme_dev * vdev Pointer to VME device struct vme_dev assigned to driver instance.

int level Interrupt priority being requested.

int statid Interrupt vector being requested.

void (*)(int, int, void *) callback Pointer to callback function called when VME interrupt/vector received.

void * priv_data Generic pointer that will be passed to the callback function.

Description

Request callback to be attached as a handler for VME interrupts with provided level and statid.

Return

Zero on success, -EINVAL on invalid vme device, level or if the function is not supported, -EBUSY if the level/statid combination is already in use. Hardware specific errors also possible.

void **vme_irq_free**(struct *vme_dev* * *vdev*, int *level*, int *statid*)
Free a VME interrupt.

Parameters

struct vme_dev * vdev Pointer to VME device struct vme_dev assigned to driver instance.

int level Interrupt priority of interrupt being freed.

int statid Interrupt vector of interrupt being freed.

Description

Remove previously attached callback from VME interrupt priority/vector.

int **vme_irq_generate**(struct *vme_dev* * *vdev*, int *level*, int *statid*)
Generate VME interrupt.

Parameters

struct vme_dev * vdev Pointer to VME device struct vme_dev assigned to driver instance.

int level Interrupt priority at which to assert the interrupt.

int statid Interrupt vector to associate with the interrupt.

Description

Generate a VME interrupt of the provided level and with the provided statid.

Return

Zero on success, -EINVAL on invalid vme device, level or if the function is not supported. Hardware specific errors also possible.

struct vme_resource * **vme_lm_request**(struct *vme_dev* * *vdev*)
Request a VME location monitor

Parameters

struct vme_dev * vdev Pointer to VME device struct vme_dev assigned to driver instance.

Description

Allocate a location monitor resource to the driver. A location monitor allows the driver to monitor accesses to a contiguous number of addresses on the VME bus.

Return

Pointer to a VME resource on success or NULL on failure.

int **vme_lm_count**(struct vme_resource * *resource*)
Determine number of VME Addresses monitored

Parameters

struct vme_resource * resource Pointer to VME location monitor resource.

Description

The number of contiguous addresses monitored is hardware dependent. Return the number of contiguous addresses monitored by the location monitor.

Return

Count of addresses monitored or -EINVAL when provided with an invalid location monitor resource.

int **vme_lm_set**(struct vme_resource * *resource*, unsigned long long *lm_base*, u32 *aspace*, u32 *cycle*)
Configure location monitor

Parameters

`struct vme_resource * resource` Pointer to VME location monitor resource.

`unsigned long long lm_base` Base address to monitor.

`u32 aspace` VME address space to monitor.

`u32 cycle` VME bus cycle type to monitor.

Description

Set the base address, address space and cycle type of accesses to be monitored by the location monitor.

Return

Zero on success, -EINVAL when provided with an invalid location monitor resource or function is not supported. Hardware specific errors may also be returned.

int **`vme_lm_get`**(struct vme_resource * *resource*, unsigned long long * *lm_base*, u32 * *aspace*, u32 * *cycle*)
Retrieve location monitor settings

Parameters

`struct vme_resource * resource` Pointer to VME location monitor resource.

`unsigned long long * lm_base` Pointer used to output the base address monitored.

`u32 * aspace` Pointer used to output the address space monitored.

`u32 * cycle` Pointer used to output the VME bus cycle type monitored.

Description

Retrieve the base address, address space and cycle type of accesses to be monitored by the location monitor.

Return

Zero on success, -EINVAL when provided with an invalid location monitor resource or function is not supported. Hardware specific errors may also be returned.

int **`vme_lm_attach`**(struct vme_resource * *resource*, int *monitor*, void (*callback) (void *, void * *data*)
Provide callback for location monitor address

Parameters

`struct vme_resource * resource` Pointer to VME location monitor resource.

`int monitor` Offset to which callback should be attached.

`void (*)(void *) callback` Pointer to callback function called when triggered.

`void * data` Generic pointer that will be passed to the callback function.

Description

Attach a callback to the specificed offset into the location monitors monitored addresses. A generic pointer is provided to allow data to be passed to the callback when called.

Return

Zero on success, -EINVAL when provided with an invalid location monitor resource or function is not supported. Hardware specific errors may also be returned.

int **`vme_lm_detach`**(struct vme_resource * *resource*, int *monitor*)
Remove callback for location monitor address

Parameters

`struct vme_resource * resource` Pointer to VME location monitor resource.

`int monitor` Offset to which callback should be removed.

Description

Remove the callback associated with the specificed offset into the location monitors monitored addresses.

Return

Zero on success, -EINVAL when provided with an invalid location monitor resource or function is not supported. Hardware specific errors may also be returned.

void **vme_lm_free**(struct vme_resource * *resource*)
Free allocated VME location monitor

Parameters

struct vme_resource * resource Pointer to VME location monitor resource.

Description

Free allocation of a VME location monitor.

WARNING: This function currently expects that any callbacks that have been attached to the location monitor have been removed.

Return

Zero on success, -EINVAL when provided with an invalid location monitor resource.

int **vme_slot_num**(struct *vme_dev* * *vdev*)
Retrieve slot ID

Parameters

struct vme_dev * vdev Pointer to VME device struct vme_dev assigned to driver instance.

Description

Retrieve the slot ID associated with the provided VME device.

Return

The slot ID on success, -EINVAL if VME bridge cannot be determined or the function is not supported. Hardware specific errors may also be returned.

int **vme_bus_num**(struct *vme_dev* * *vdev*)
Retrieve bus number

Parameters

struct vme_dev * vdev Pointer to VME device struct vme_dev assigned to driver instance.

Description

Retrieve the bus enumeration associated with the provided VME device.

Return

The bus number on success, -EINVAL if VME bridge cannot be determined.

int **vme_register_driver**(struct *vme_driver* * *drv*, unsigned int *ndevs*)
Register a VME driver

Parameters

struct vme_driver * drv Pointer to VME driver structure to register.

unsigned int ndevs Maximum number of devices to allow to be enumerated.

Description

Register a VME device driver with the VME subsystem.

Return

Zero on success, error value on registration failure.

void **`vme_unregister_driver`**(struct *vme_driver* * *drv*)
Unregister a VME driver

Parameters

`struct vme_driver * drv` Pointer to VME driver structure to unregister.

Description

Unregister a VME device driver from the VME subsystem.

CHAPTER

THIRTY

LINUX 802.11 DRIVER DEVELOPER'S GUIDE

Introduction

Explaining wireless 802.11 networking in the Linux kernel

Copyright 2007-2009 Johannes Berg

These books attempt to give a description of the various subsystems that play a role in 802.11 wireless networking in Linux. Since these books are for kernel developers they attempts to document the structures and functions used in the kernel as well as giving a higher-level overview.

The reader is expected to be familiar with the 802.11 standard as published by the IEEE in 802.11-2007 (or possibly later versions). References to this standard will be given as "802.11-2007 8.1.5".

cfg80211 subsystem

cfg80211 is the configuration API for 802.11 devices in Linux. It bridges userspace and drivers, and offers some utility functionality associated with 802.11. cfg80211 must, directly or indirectly via mac80211, be used by all modern wireless drivers in Linux, so that they offer a consistent API through nl80211. For backward compatibility, cfg80211 also offers wireless extensions to userspace, but hides them from drivers completely.

Additionally, cfg80211 contains code to help enforce regulatory spectrum use restrictions.

Device registration

In order for a driver to use cfg80211, it must register the hardware device with cfg80211. This happens through a number of hardware capability structs described below.

The fundamental structure for each device is the 'wiphy', of which each instance describes a physical wireless device connected to the system. Each such wiphy can have zero, one, or many virtual interfaces associated with it, which need to be identified as such by pointing the network interface's **ieee80211_ptr** pointer to a *`struct wireless_dev`* which further describes the wireless part of the interface, normally this struct is embedded in the network interface's private data area. Drivers can optionally allow creating or destroying virtual interfaces on the fly, but without at least one or the ability to create some the wireless device isn't useful.

Each wiphy structure contains device capability information, and also has a pointer to the various operations the driver offers. The definitions and structures here describe these capabilities in detail.

enum **`ieee80211_channel_flags`**
channel flags

Constants

`IEEE80211_CHAN_DISABLED` This channel is disabled.

`IEEE80211_CHAN_NO_IR` do not initiate radiation, this includes sending probe requests or beaconing.

IEEE80211_CHAN_RADAR Radar detection is required on this channel.

IEEE80211_CHAN_NO_HT40PLUS extension channel above this channel is not permitted.

IEEE80211_CHAN_NO_HT40MINUS extension channel below this channel is not permitted.

IEEE80211_CHAN_NO_OFDM OFDM is not allowed on this channel.

IEEE80211_CHAN_NO_80MHZ If the driver supports 80 MHz on the band, this flag indicates that an 80 MHz channel cannot use this channel as the control or any of the secondary channels. This may be due to the driver or due to regulatory bandwidth restrictions.

IEEE80211_CHAN_NO_160MHZ If the driver supports 160 MHz on the band, this flag indicates that an 160 MHz channel cannot use this channel as the control or any of the secondary channels. This may be due to the driver or due to regulatory bandwidth restrictions.

IEEE80211_CHAN_INDOOR_ONLY see NL80211_FREQUENCY_ATTR_INDOOR_ONLY

IEEE80211_CHAN_IR_CONCURRENT see NL80211_FREQUENCY_ATTR_IR_CONCURRENT

IEEE80211_CHAN_NO_20MHZ 20 MHz bandwidth is not permitted on this channel.

IEEE80211_CHAN_NO_10MHZ 10 MHz bandwidth is not permitted on this channel.

Description

Channel flags set by the regulatory control code.

struct **ieee80211_channel**
channel definition

Definition

```
struct ieee80211_channel {
  enum nl80211_band band;
  u16 center_freq;
  u16 hw_value;
  u32 flags;
  int max_antenna_gain;
  int max_power;
  int max_reg_power;
  bool beacon_found;
  u32 orig_flags;
  int orig_mag, orig_mpwr;
  enum nl80211_dfs_state dfs_state;
  unsigned long dfs_state_entered;
  unsigned int dfs_cac_ms;
};
```

Members

band band this channel belongs to.

center_freq center frequency in MHz

hw_value hardware-specific value for the channel

flags channel flags from *enum ieee80211_channel_flags*.

max_antenna_gain maximum antenna gain in dBi

max_power maximum transmission power (in dBm)

max_reg_power maximum regulatory transmission power (in dBm)

beacon_found helper to regulatory code to indicate when a beacon has been found on this channel. Use regulatory_hint_found_beacon() to enable this, this is useful only on 5 GHz band.

orig_flags channel flags at registration time, used by regulatory code to support devices with additional restrictions

orig_mag internal use

orig_mpwr internal use

dfs_state current state of this channel. Only relevant if radar is required on this channel.

dfs_state_entered timestamp (jiffies) when the dfs state was entered.

dfs_cac_ms DFS CAC time in milliseconds, this is valid for DFS channels.

Description

This structure describes a single channel for use with cfg80211.

enum **ieee80211_rate_flags**
rate flags

Constants

IEEE80211_RATE_SHORT_PREAMBLE Hardware can send with short preamble on this bitrate; only relevant in 2.4GHz band and with CCK rates.

IEEE80211_RATE_MANDATORY_A This bitrate is a mandatory rate when used with 802.11a (on the 5 GHz band); filled by the core code when registering the wiphy.

IEEE80211_RATE_MANDATORY_B This bitrate is a mandatory rate when used with 802.11b (on the 2.4 GHz band); filled by the core code when registering the wiphy.

IEEE80211_RATE_MANDATORY_G This bitrate is a mandatory rate when used with 802.11g (on the 2.4 GHz band); filled by the core code when registering the wiphy.

IEEE80211_RATE_ERP_G This is an ERP rate in 802.11g mode.

IEEE80211_RATE_SUPPORTS_5MHZ Rate can be used in 5 MHz mode

IEEE80211_RATE_SUPPORTS_10MHZ Rate can be used in 10 MHz mode

Description

Hardware/specification flags for rates. These are structured in a way that allows using the same bitrate structure for different bands/PHY modes.

struct **ieee80211_rate**
bitrate definition

Definition

```
struct ieee80211_rate {
  u32 flags;
  u16 bitrate;
  u16 hw_value, hw_value_short;
};
```

Members

flags rate-specific flags

bitrate bitrate in units of 100 Kbps

hw_value driver/hardware value for this rate

hw_value_short driver/hardware value for this rate when short preamble is used

Description

This structure describes a bitrate that an 802.11 PHY can operate with. The two values **hw_value** and **hw_value_short** are only for driver use when pointers to this structure are passed around.

struct **ieee80211_sta_ht_cap**
STA's HT capabilities

Definition

```
struct ieee80211_sta_ht_cap {
  u16 cap;
  bool ht_supported;
  u8 ampdu_factor;
  u8 ampdu_density;
  struct ieee80211_mcs_info mcs;
};
```

Members

cap HT capabilities map as described in 802.11n spec

ht_supported is HT supported by the STA

ampdu_factor Maximum A-MPDU length factor

ampdu_density Minimum A-MPDU spacing

mcs Supported MCS rates

Description

This structure describes most essential parameters needed to describe 802.11n HT capabilities for an STA.

struct **ieee80211_supported_band**
frequency band definition

Definition

```
struct ieee80211_supported_band {
  struct ieee80211_channel *channels;
  struct ieee80211_rate *bitrates;
  enum nl80211_band band;
  int n_channels;
  int n_bitrates;
  struct ieee80211_sta_ht_cap ht_cap;
  struct ieee80211_sta_vht_cap vht_cap;
};
```

Members

channels Array of channels the hardware can operate in in this band.

bitrates Array of bitrates the hardware can operate with in this band. Must be sorted to give a valid "supported rates" IE, i.e. CCK rates first, then OFDM.

band the band this structure represents

n_channels Number of channels in **channels**

n_bitrates Number of bitrates in **bitrates**

ht_cap HT capabilities in this band

vht_cap VHT capabilities in this band

Description

This structure describes a frequency band a wiphy is able to operate in.

enum **cfg80211_signal_type**
signal type

Constants

CFG80211_SIGNAL_TYPE_NONE no signal strength information available

CFG80211_SIGNAL_TYPE_MBM signal strength in mBm (100*dBm)

CFG80211_SIGNAL_TYPE_UNSPEC signal strength, increasing from 0 through 100

enum **wiphy_params_flags**
set_wiphy_params bitfield values

Constants

WIPHY_PARAM_RETRY_SHORT wiphy->retry_short has changed

WIPHY_PARAM_RETRY_LONG wiphy->retry_long has changed

WIPHY_PARAM_FRAG_THRESHOLD wiphy->frag_threshold has changed

WIPHY_PARAM_RTS_THRESHOLD wiphy->rts_threshold has changed

WIPHY_PARAM_COVERAGE_CLASS coverage class changed

WIPHY_PARAM_DYN_ACK dynack has been enabled

enum **wiphy_flags**
wiphy capability flags

Constants

WIPHY_FLAG_NETNS_OK if not set, do not allow changing the netns of this wiphy at all

WIPHY_FLAG_PS_ON_BY_DEFAULT if set to true, powersave will be enabled by default – this flag will be set depending on the kernel's default on *wiphy_new()*, but can be changed by the driver if it has a good reason to override the default

WIPHY_FLAG_4ADDR_AP supports 4addr mode even on AP (with a single station on a VLAN interface)

WIPHY_FLAG_4ADDR_STATION supports 4addr mode even as a station

WIPHY_FLAG_CONTROL_PORT_PROTOCOL This device supports setting the control port protocol ethertype. The device also honours the control_port_no_encrypt flag.

WIPHY_FLAG_IBSS_RSN The device supports IBSS RSN.

WIPHY_FLAG_MESH_AUTH The device supports mesh authentication by routing auth frames to userspace. See **NL80211_MESH_SETUP_USERSPACE_AUTH**.

WIPHY_FLAG_SUPPORTS_FW_ROAM The device supports roaming feature in the firmware.

WIPHY_FLAG_AP_UAPSD The device supports uapsd on AP.

WIPHY_FLAG_SUPPORTS_TDLS The device supports TDLS (802.11z) operation.

WIPHY_FLAG_TDLS_EXTERNAL_SETUP The device does not handle TDLS (802.11z) link setup/discovery operations internally. Setup, discovery and teardown packets should be sent through the **NL80211_CMD_TDLS_MGMT** command. When this flag is not set, **NL80211_CMD_TDLS_OPER** should be used for asking the driver/firmware to perform a TDLS operation.

WIPHY_FLAG_HAVE_AP_SME device integrates AP SME

WIPHY_FLAG_REPORTS_OBSS the device will report beacons from other BSSes when there are virtual interfaces in AP mode by calling `cfg80211_report_obss_beacon()`.

WIPHY_FLAG_AP_PROBE_RESP_OFFLOAD When operating as an AP, the device responds to probe-requests in hardware.

WIPHY_FLAG_OFFCHAN_TX Device supports direct off-channel TX.

WIPHY_FLAG_HAS_REMAIN_ON_CHANNEL Device supports remain-on-channel call.

WIPHY_FLAG_SUPPORTS_5_10_MHZ Device supports 5 MHz and 10 MHz channels.

WIPHY_FLAG_HAS_CHANNEL_SWITCH Device supports channel switch in beaconing mode (AP, IBSS, Mesh, ...).

WIPHY_FLAG_HAS_STATIC_WEP The device supports static WEP key installation before connection.

struct **wiphy**
wireless hardware description

Definition

```
struct wiphy {
  u8 perm_addr[ETH_ALEN];
  u8 addr_mask[ETH_ALEN];
  struct mac_address *addresses;
  const struct ieee80211_txrx_stypes *mgmt_stypes;
  const struct ieee80211_iface_combination *iface_combinations;
  int n_iface_combinations;
  u16 software_iftypes;
  u16 n_addresses;
  u16 interface_modes;
  u16 max_acl_mac_addrs;
  u32 flags, regulatory_flags, features;
  u8 ext_features[DIV_ROUND_UP(NUM_NL80211_EXT_FEATURES, 8)];
  u32 ap_sme_capa;
  enum cfg80211_signal_type signal_type;
  int bss_priv_size;
  u8 max_scan_ssids;
  u8 max_sched_scan_reqs;
  u8 max_sched_scan_ssids;
  u8 max_match_sets;
  u16 max_scan_ie_len;
  u16 max_sched_scan_ie_len;
  u32 max_sched_scan_plans;
  u32 max_sched_scan_plan_interval;
  u32 max_sched_scan_plan_iterations;
  int n_cipher_suites;
  const u32 *cipher_suites;
  u8 retry_short;
  u8 retry_long;
  u32 frag_threshold;
  u32 rts_threshold;
  u8 coverage_class;
  char fw_version[ETHTOOL_FWVERS_LEN];
  u32 hw_version;
#ifdef CONFIG_PM;
  const struct wiphy_wowlan_support *wowlan;
  struct cfg80211_wowlan *wowlan_config;
#endif;
  u16 max_remain_on_channel_duration;
  u8 max_num_pmkids;
  u32 available_antennas_tx;
  u32 available_antennas_rx;
  u32 probe_resp_offload;
  const u8 *extended_capabilities, *extended_capabilities_mask;
  u8 extended_capabilities_len;
  const struct wiphy_iftype_ext_capab *iftype_ext_capab;
  unsigned int num_iftype_ext_capab;
  const void *privid;
  struct ieee80211_supported_band *bands[NUM_NL80211_BANDS];
  void (*reg_notifier)(struct wiphy *wiphy, struct regulatory_request *request);
  const struct ieee80211_regdomain __rcu *regd;
  struct device dev;
  bool registered;
  struct dentry *debugfsdir;
  const struct ieee80211_ht_cap *ht_capa_mod_mask;
  const struct ieee80211_vht_cap *vht_capa_mod_mask;
  struct list_head wdev_list;
  possible_net_t _net;
#ifdef CONFIG_CFG80211_WEXT;
  const struct iw_handler_def *wext;
#endif;
```

```
  const struct wiphy_coalesce_support *coalesce;
  const struct wiphy_vendor_command *vendor_commands;
  const struct nl80211_vendor_cmd_info *vendor_events;
  int n_vendor_commands, n_vendor_events;
  u16 max_ap_assoc_sta;
  u8 max_num_csa_counters;
  u8 max_adj_channel_rssi_comp;
  u32 bss_select_support;
  u64 cookie_counter;
  u8 nan_supported_bands;
  char priv[0] ;
};
```

Members

perm_addr permanent MAC address of this device

addr_mask If the device supports multiple MAC addresses by masking, set this to a mask with variable bits set to 1, e.g. if the last four bits are variable then set it to 00-00-00-00-00-0f. The actual variable bits shall be determined by the interfaces added, with interfaces not matching the mask being rejected to be brought up.

addresses If the device has more than one address, set this pointer to a list of addresses (6 bytes each). The first one will be used by default for perm_addr. In this case, the mask should be set to all-zeroes. In this case it is assumed that the device can handle the same number of arbitrary MAC addresses.

mgmt_stypes bitmasks of frame subtypes that can be subscribed to or transmitted through nl80211, points to an array indexed by interface type

iface_combinations Valid interface combinations array, should not list single interface types.

n_iface_combinations number of entries in **iface_combinations** array.

software_iftypes bitmask of software interface types, these are not subject to any restrictions since they are purely managed in SW.

n_addresses number of addresses in **addresses**.

interface_modes bitmask of interfaces types valid for this wiphy, must be set by driver

max_acl_mac_addrs Maximum number of MAC addresses that the device supports for ACL.

flags wiphy flags, see *enum wiphy_flags*

regulatory_flags wiphy regulatory flags, see enum `ieee80211_regulatory_flags`

features features advertised to nl80211, see enum `nl80211_feature_flags`.

ext_features extended features advertised to nl80211, see enum `nl80211_ext_feature_index`.

ap_sme_capa AP SME capabilities, flags from enum `nl80211_ap_sme_features`.

signal_type signal type reported in *struct cfg80211_bss*.

bss_priv_size each BSS struct has private data allocated with it, this variable determines its size

max_scan_ssids maximum number of SSIDs the device can scan for in any given scan

max_sched_scan_reqs maximum number of scheduled scan requests that the device can run concurrently.

max_sched_scan_ssids maximum number of SSIDs the device can scan for in any given scheduled scan

max_match_sets maximum number of match sets the device can handle when performing a scheduled scan, 0 if filtering is not supported.

max_scan_ie_len maximum length of user-controlled IEs device can add to probe request frames transmitted during a scan, must not include fixed IEs like supported rates

max_sched_scan_ie_len same as max_scan_ie_len, but for scheduled scans

max_sched_scan_plans maximum number of scan plans (scan interval and number of iterations) for scheduled scan supported by the device.

max_sched_scan_plan_interval maximum interval (in seconds) for a single scan plan supported by the device.

max_sched_scan_plan_iterations maximum number of iterations for a single scan plan supported by the device.

n_cipher_suites number of supported cipher suites

cipher_suites supported cipher suites

retry_short Retry limit for short frames (dot11ShortRetryLimit)

retry_long Retry limit for long frames (dot11LongRetryLimit)

frag_threshold Fragmentation threshold (dot11FragmentationThreshold); -1 = fragmentation disabled, only odd values >= 256 used

rts_threshold RTS threshold (dot11RTSThreshold); -1 = RTS/CTS disabled

coverage_class current coverage class

fw_version firmware version for ethtool reporting

hw_version hardware version for ethtool reporting

wowlan WoWLAN support information

wowlan_config current WoWLAN configuration; this should usually not be used since access to it is necessarily racy, use the parameter passed to the `suspend()` operation instead.

max_remain_on_channel_duration Maximum time a remain-on-channel operation may request, if implemented.

max_num_pmkids maximum number of PMKIDs supported by device

available_antennas_tx bitmap of antennas which are available to be configured as TX antennas. Antenna configuration commands will be rejected unless this or **available_antennas_rx** is set.

available_antennas_rx bitmap of antennas which are available to be configured as RX antennas. Antenna configuration commands will be rejected unless this or **available_antennas_tx** is set.

probe_resp_offload Bitmap of supported protocols for probe response offloading. See `enum nl80211_probe_resp_offload_support_attr`. Only valid when the wiphy flag **WIPHY_FLAG_AP_PROBE_RESP_OFFLOAD** is set.

extended_capabilities extended capabilities supported by the driver, additional capabilities might be supported by userspace; these are the 802.11 extended capabilities ("Extended Capabilities element") and are in the same format as in the information element. See 802.11-2012 8.4.2.29 for the defined fields. These are the default extended capabilities to be used if the capabilities are not specified for a specific interface type in iftype_ext_capab.

extended_capabilities_mask mask of the valid values

extended_capabilities_len length of the extended capabilities

iftype_ext_capab array of extended capabilities per interface type

num_iftype_ext_capab number of interface types for which extended capabilities are specified separately.

privid a pointer that drivers can use to identify if an arbitrary wiphy is theirs, e.g. in global notifiers

bands information about bands/channels supported by this device

reg_notifier the driver's regulatory notification callback, note that if your driver uses *wiphy_apply_custom_regulatory()* the reg_notifier's request can be passed as NULL

regd the driver's regulatory domain, if one was requested via the *regulatory_hint()* API. This can be used by the driver on the reg_notifier() if it chooses to ignore future regulatory domain changes caused by other drivers.

dev (virtual) struct device for this wiphy

registered helps synchronize suspend/resume with wiphy unregister

debugfsdir debugfs directory used for this wiphy, will be renamed automatically on wiphy renames

ht_capa_mod_mask Specify what ht_cap values can be over-ridden. If null, then none can be over-ridden.

vht_capa_mod_mask Specify what VHT capabilities can be over-ridden. If null, then none can be over-ridden.

wdev_list the list of associated (virtual) interfaces; this list must not be modified by the driver, but can be read with RTNL/RCU protection.

_net the network namespace this wiphy currently lives in

wext wireless extension handlers

coalesce packet coalescing support information

vendor_commands array of vendor commands supported by the hardware

vendor_events array of vendor events supported by the hardware

n_vendor_commands number of vendor commands

n_vendor_events number of vendor events

max_ap_assoc_sta maximum number of associated stations supported in AP mode (including P2P GO) or 0 to indicate no such limit is advertised. The driver is allowed to advertise a theoretical limit that it can reach in some cases, but may not always reach.

max_num_csa_counters Number of supported csa_counters in beacons and probe responses. This value should be set if the driver wishes to limit the number of csa counters. Default (0) means infinite.

max_adj_channel_rssi_comp max offset of between the channel on which the frame was sent and the channel on which the frame was heard for which the reported rssi is still valid. If a driver is able to compensate the low rssi when a frame is heard on different channel, then it should set this variable to the maximal offset for which it can compensate. This value should be set in MHz.

bss_select_support bitmask indicating the BSS selection criteria supported by the driver in the .:c:func:*connect()* callback. The bit position maps to the attribute indices defined in enum nl80211_bss_select_attr.

cookie_counter unique generic cookie counter, used to identify objects.

nan_supported_bands bands supported by the device in NAN mode, a bitmap of enum nl80211_band values. For instance, for NL80211_BAND_2GHZ, bit 0 would be set (i.e. BIT(NL80211_BAND_2GHZ)).

priv driver private data (sized according to *wiphy_new()* parameter)

struct **wireless_dev**
wireless device state

Definition

```
struct wireless_dev {
  struct wiphy *wiphy;
  enum nl80211_iftype iftype;
  struct list_head list;
  struct net_device *netdev;
  u32 identifier;
  struct list_head mgmt_registrations;
  spinlock_t mgmt_registrations_lock;
  struct mutex mtx;
  bool use_4addr, is_running;
  u8 address[ETH_ALEN] ;
  u8 ssid[IEEE80211_MAX_SSID_LEN];
  u8 ssid_len, mesh_id_len, mesh_id_up_len;
  struct cfg80211_conn *conn;
  struct cfg80211_cached_keys *connect_keys;
  enum ieee80211_bss_type conn_bss_type;
  u32 conn_owner_nlportid;
  struct work_struct disconnect_wk;
  u8 disconnect_bssid[ETH_ALEN];
  struct list_head event_list;
  spinlock_t event_lock;
  struct cfg80211_internal_bss *current_bss;
  struct cfg80211_chan_def preset_chandef;
  struct cfg80211_chan_def chandef;
  bool ibss_fixed;
  bool ibss_dfs_possible;
  bool ps;
  int ps_timeout;
  int beacon_interval;
  u32 ap_unexpected_nlportid;
  u32 owner_nlportid;
  bool nl_owner_dead;
  bool cac_started;
  unsigned long cac_start_time;
  unsigned int cac_time_ms;
#ifdef CONFIG_CFG80211_WEXT;
  struct {
    struct cfg80211_ibss_params ibss;
    struct cfg80211_connect_params connect;
    struct cfg80211_cached_keys *keys;
    const u8 *ie;
    size_t ie_len;
    u8 bssid[ETH_ALEN], prev_bssid[ETH_ALEN];
    u8 ssid[IEEE80211_MAX_SSID_LEN];
    s8 default_key, default_mgmt_key;
    bool prev_bssid_valid;
  } wext;
#endif;
  struct cfg80211_cqm_config *cqm_config;
};
```

Members

wiphy pointer to hardware description

iftype interface type

list (private) Used to collect the interfaces

netdev (private) Used to reference back to the netdev, may be NULL

identifier (private) Identifier used in nl80211 to identify this wireless device if it has no netdev

mgmt_registrations list of registrations for management frames

mgmt_registrations_lock lock for the list

mtx mutex used to lock data in this struct, may be used by drivers and some API functions require it held

use_4addr indicates 4addr mode is used on this interface, must be set by driver (if supported) on add_interface BEFORE registering the netdev and may otherwise be used by driver read-only, will be update by cfg80211 on change_interface

is_running true if this is a non-netdev device that has been started, e.g. the P2P Device.

address The address for this device, valid only if **netdev** is NULL

ssid (private) Used by the internal configuration code

ssid_len (private) Used by the internal configuration code

mesh_id_len (private) Used by the internal configuration code

mesh_id_up_len (private) Used by the internal configuration code

conn (private) cfg80211 software SME connection state machine data

connect_keys (private) keys to set after connection is established

conn_bss_type connecting/connected BSS type

conn_owner_nlportid (private) connection owner socket port ID

disconnect_wk (private) auto-disconnect work

disconnect_bssid (private) the BSSID to use for auto-disconnect

event_list (private) list for internal event processing

event_lock (private) lock for event list

current_bss (private) Used by the internal configuration code

preset_chandef (private) Used by the internal configuration code to track the channel to be used for AP later

chandef (private) Used by the internal configuration code to track the user-set channel definition.

ibss_fixed (private) IBSS is using fixed BSSID

ibss_dfs_possible (private) IBSS may change to a DFS channel

ps powersave mode is enabled

ps_timeout dynamic powersave timeout

beacon_interval beacon interval used on this device for transmitting beacons, 0 when not valid

ap_unexpected_nlportid (private) netlink port ID of application registered for unexpected class 3 frames (AP mode)

owner_nlportid (private) owner socket port ID

nl_owner_dead (private) owner socket went away

cac_started true if DFS channel availability check has been started

cac_start_time timestamp (jiffies) when the dfs state was entered.

cac_time_ms CAC time in ms

wext (private) Used by the internal wireless extensions compat code

cqm_config (private) nl80211 RSSI monitor state

Description

For netdevs, this structure must be allocated by the driver that uses the ieee80211_ptr field in struct net_device (this is intentional so it can be allocated along with the netdev.) It need not be registered then as netdev registration will be intercepted by cfg80211 to see the new wireless device.

For non-netdev uses, it must also be allocated by the driver in response to the cfg80211 callbacks that require it, as there's no netdev registration in that case it may not be allocated outside of callback operations that return it.

struct *wiphy* * **wiphy_new**(const struct *cfg80211_ops* * *ops*, int *sizeof_priv*)
create a new wiphy for use with cfg80211

Parameters

const struct cfg80211_ops * ops The configuration operations for this device

`int sizeof_priv` The size of the private area to allocate

Description

Create a new wiphy and associate the given operations with it. **sizeof_priv** bytes are allocated for private use.

Return

A pointer to the new wiphy. This pointer must be assigned to each netdev's ieee80211_ptr for proper operation.

void **`wiphy_read_of_freq_limits`**(struct *wiphy* * *wiphy*)
    read frequency limits from device tree

Parameters

`struct wiphy * wiphy` the wireless device to get extra limits for

Description

Some devices may have extra limitations specified in DT. This may be useful for chipsets that normally support more bands but are limited due to board design (e.g. by antennas or external power amplifier).

This function reads info from DT and uses it to *modify* channels (disable unavailable ones). It's usually a *bad* idea to use it in drivers with shared channel data as DT limitations are device specific. You should make sure to call it only if channels in wiphy are copied and can be modified without affecting other devices.

As this function access device node it has to be called after set_wiphy_dev. It also modifies channels so they have to be set first. If using this helper, call it before *`wiphy_register()`*.

int **`wiphy_register`**(struct *wiphy* * *wiphy*)
    register a wiphy with cfg80211

Parameters

`struct wiphy * wiphy` The wiphy to register.

Return

A non-negative wiphy index or a negative error code.

void **`wiphy_unregister`**(struct *wiphy* * *wiphy*)
    deregister a wiphy from cfg80211

Parameters

`struct wiphy * wiphy` The wiphy to unregister.

Description

After this call, no more requests can be made with this priv pointer, but the call may sleep to wait for an outstanding request that is being handled.

void **`wiphy_free`**(struct *wiphy* * *wiphy*)
    free wiphy

Parameters

`struct wiphy * wiphy` The wiphy to free

const char * **`wiphy_name`**(const struct *wiphy* * *wiphy*)
    get wiphy name

Parameters

`const struct wiphy * wiphy` The wiphy whose name to return

Return

The name of **wiphy**.

struct *device* * **wiphy_dev**(struct *wiphy* * *wiphy*)
get wiphy dev pointer

Parameters

struct wiphy * wiphy The wiphy whose device struct to look up

Return

The dev of **wiphy**.

void * **wiphy_priv**(struct *wiphy* * *wiphy*)
return priv from wiphy

Parameters

struct wiphy * wiphy the wiphy whose priv pointer to return

Return

The priv of **wiphy**.

struct *wiphy* * **priv_to_wiphy**(void * *priv*)
return the wiphy containing the priv

Parameters

void * priv a pointer previously returned by wiphy_priv

Return

The wiphy of **priv**.

void **set_wiphy_dev**(struct *wiphy* * *wiphy*, struct *device* * *dev*)
set device pointer for wiphy

Parameters

struct wiphy * wiphy The wiphy whose device to bind

struct device * dev The device to parent it to

void * **wdev_priv**(struct *wireless_dev* * *wdev*)
return wiphy priv from wireless_dev

Parameters

struct wireless_dev * wdev The wireless device whose wiphy's priv pointer to return

Return

The wiphy priv of **wdev**.

struct **ieee80211_iface_limit**
limit on certain interface types

Definition

```
struct ieee80211_iface_limit {
  u16 max;
  u16 types;
};
```

Members

max maximum number of interfaces of these types

types interface types (bits)

struct **ieee80211_iface_combination**
possible interface combination

Definition

```
struct ieee80211_iface_combination {
  const struct ieee80211_iface_limit *limits;
  u32 num_different_channels;
  u16 max_interfaces;
  u8 n_limits;
  bool beacon_int_infra_match;
  u8 radar_detect_widths;
  u8 radar_detect_regions;
  u32 beacon_int_min_gcd;
};
```

Members

`limits` limits for the given interface types

`num_different_channels` can use up to this many different channels

`max_interfaces` maximum number of interfaces in total allowed in this group

`n_limits` number of limitations

`beacon_int_infra_match` In this combination, the beacon intervals between infrastructure and AP types must match. This is required only in special cases.

`radar_detect_widths` bitmap of channel widths supported for radar detection

`radar_detect_regions` bitmap of regions supported for radar detection

`beacon_int_min_gcd` This interface combination supports different beacon intervals.

= 0 all beacon intervals for different interface must be same.

> 0 any beacon interval for the interface part of this combination AND GCD of all beacon intervals from beaconing interfaces of this combination must be greater or equal to this value.

Description

With this structure the driver can describe which interface combinations it supports concurrently.

Examples

1. Allow #STA <= 1, #AP <= 1, matching BI, channels = 1, 2 total:

```
    struct ieee80211_iface_limit limits1[] = {
            { .max = 1, .types = BIT(NL80211_IFTYPE_STATION), },
            { .max = 1, .types = BIT(NL80211_IFTYPE_AP}, },
    };
    struct ieee80211_iface_combination combination1 = {
            .limits = limits1,
            .n_limits = ARRAY_SIZE(limits1),
            .max_interfaces = 2,
            .beacon_int_infra_match = true,
    };
```

2. Allow #{AP, P2P-GO} <= 8, channels = 1, 8 total:

```
    struct ieee80211_iface_limit limits2[] = {
            { .max = 8, .types = BIT(NL80211_IFTYPE_AP) |
                                 BIT(NL80211_IFTYPE_P2P_GO), },
    };
    struct ieee80211_iface_combination combination2 = {
            .limits = limits2,
            .n_limits = ARRAY_SIZE(limits2),
            .max_interfaces = 8,
            .num_different_channels = 1,
    };
```

3. Allow #STA <= 1, #{P2P-client,P2P-GO} <= 3 on two channels, 4 total.

   This allows for an infrastructure connection and three P2P connections.

```
    struct ieee80211_iface_limit limits3[] = {
            { .max = 1, .types = BIT(NL80211_IFTYPE_STATION), },
            { .max = 3, .types = BIT(NL80211_IFTYPE_P2P_GO) |
                                 BIT(NL80211_IFTYPE_P2P_CLIENT), },
    };
    struct ieee80211_iface_combination combination3 = {
            .limits = limits3,
            .n_limits = ARRAY_SIZE(limits3),
            .max_interfaces = 4,
            .num_different_channels = 2,
    };
```

int **cfg80211_check_combinations**(struct *wiphy* * *wiphy*, struct iface_combination_params * *params*)
check interface combinations

Parameters

struct wiphy * wiphy the wiphy

struct iface_combination_params * params the interface combinations parameter

Description

This function can be called by the driver to check whether a combination of interfaces and their types are allowed according to the interface combinations.

Actions and configuration

Each wireless device and each virtual interface offer a set of configuration operations and other actions that are invoked by userspace. Each of these actions is described in the operations structure, and the parameters these operations use are described separately.

Additionally, some operations are asynchronous and expect to get status information via some functions that drivers need to call.

Scanning and BSS list handling with its associated functionality is described in a separate chapter.

struct **cfg80211_ops**
backend description for wireless configuration

Definition

```
struct cfg80211_ops {
  int (*suspend)(struct wiphy *wiphy, struct cfg80211_wowlan *wow);
  int (*resume)(struct wiphy *wiphy);
  void (*set_wakeup)(struct wiphy *wiphy, bool enabled);
  struct wireless_dev * (*add_virtual_intf)(struct wiphy *wiphy,const char *name,unsigned char name_assi
  int (*del_virtual_intf)(struct wiphy *wiphy, struct wireless_dev *wdev);
  int (*change_virtual_intf)(struct wiphy *wiphy,struct net_device *dev,enum nl80211_iftype type, struct
  int (*add_key)(struct wiphy *wiphy, struct net_device *netdev,u8 key_index, bool pairwise, const u8 *m
  int (*get_key)(struct wiphy *wiphy, struct net_device *netdev,u8 key_index, bool pairwise, const u8 *m
  int (*del_key)(struct wiphy *wiphy, struct net_device *netdev, u8 key_index, bool pairwise, const u8 *
  int (*set_default_key)(struct wiphy *wiphy,struct net_device *netdev, u8 key_index, bool unicast, bool
  int (*set_default_mgmt_key)(struct wiphy *wiphy,struct net_device *netdev, u8 key_index);
  int (*start_ap)(struct wiphy *wiphy, struct net_device *dev, struct cfg80211_ap_settings *settings);
  int (*change_beacon)(struct wiphy *wiphy, struct net_device *dev, struct cfg80211_beacon_data *info);
  int (*stop_ap)(struct wiphy *wiphy, struct net_device *dev);
  int (*add_station)(struct wiphy *wiphy, struct net_device *dev,const u8 *mac, struct station_parameter
  int (*del_station)(struct wiphy *wiphy, struct net_device *dev, struct station_del_parameters *params)
  int (*change_station)(struct wiphy *wiphy, struct net_device *dev,const u8 *mac, struct station_parame
  int (*get_station)(struct wiphy *wiphy, struct net_device *dev, const u8 *mac, struct station_info *si
  int (*dump_station)(struct wiphy *wiphy, struct net_device *dev, int idx, u8 *mac, struct station_info
  int (*add_mpath)(struct wiphy *wiphy, struct net_device *dev, const u8 *dst, const u8 *next_hop);
  int (*del_mpath)(struct wiphy *wiphy, struct net_device *dev, const u8 *dst);
  int (*change_mpath)(struct wiphy *wiphy, struct net_device *dev, const u8 *dst, const u8 *next_hop);
  int (*get_mpath)(struct wiphy *wiphy, struct net_device *dev, u8 *dst, u8 *next_hop, struct mpath_info
  int (*dump_mpath)(struct wiphy *wiphy, struct net_device *dev,int idx, u8 *dst, u8 *next_hop, struct m
  int (*get_mpp)(struct wiphy *wiphy, struct net_device *dev, u8 *dst, u8 *mpp, struct mpath_info *pinfo
  int (*dump_mpp)(struct wiphy *wiphy, struct net_device *dev,int idx, u8 *dst, u8 *mpp, struct mpath_in
  int (*get_mesh_config)(struct wiphy *wiphy,struct net_device *dev, struct mesh_config *conf);
  int (*update_mesh_config)(struct wiphy *wiphy,struct net_device *dev, u32 mask, const struct mesh_conf
  int (*join_mesh)(struct wiphy *wiphy, struct net_device *dev,const struct mesh_config *conf, const str
  int (*leave_mesh)(struct wiphy *wiphy, struct net_device *dev);
  int (*join_ocb)(struct wiphy *wiphy, struct net_device *dev, struct ocb_setup *setup);
  int (*leave_ocb)(struct wiphy *wiphy, struct net_device *dev);
  int (*change_bss)(struct wiphy *wiphy, struct net_device *dev, struct bss_parameters *params);
  int (*set_txq_params)(struct wiphy *wiphy, struct net_device *dev, struct ieee80211_txq_params *params
  int (*libertas_set_mesh_channel)(struct wiphy *wiphy,struct net_device *dev, struct ieee80211_channel 
  int (*set_monitor_channel)(struct wiphy *wiphy, struct cfg80211_chan_def *chandef);
  int (*scan)(struct wiphy *wiphy, struct cfg80211_scan_request *request);
  void (*abort_scan)(struct wiphy *wiphy, struct wireless_dev *wdev);
  int (*auth)(struct wiphy *wiphy, struct net_device *dev, struct cfg80211_auth_request *req);
  int (*assoc)(struct wiphy *wiphy, struct net_device *dev, struct cfg80211_assoc_request *req);
  int (*deauth)(struct wiphy *wiphy, struct net_device *dev, struct cfg80211_deauth_request *req);
  int (*disassoc)(struct wiphy *wiphy, struct net_device *dev, struct cfg80211_disassoc_request *req);
  int (*connect)(struct wiphy *wiphy, struct net_device *dev, struct cfg80211_connect_params *sme);
  int (*update_connect_params)(struct wiphy *wiphy,struct net_device *dev,struct cfg80211_connect_params
  int (*disconnect)(struct wiphy *wiphy, struct net_device *dev, u16 reason_code);
  int (*join_ibss)(struct wiphy *wiphy, struct net_device *dev, struct cfg80211_ibss_params *params);
  int (*leave_ibss)(struct wiphy *wiphy, struct net_device *dev);
  int (*set_mcast_rate)(struct wiphy *wiphy, struct net_device *dev, int rate[NUM_NL80211_BANDS]);
  int (*set_wiphy_params)(struct wiphy *wiphy, u32 changed);
  int (*set_tx_power)(struct wiphy *wiphy, struct wireless_dev *wdev, enum nl80211_tx_power_setting type
  int (*get_tx_power)(struct wiphy *wiphy, struct wireless_dev *wdev, int *dbm);
  int (*set_wds_peer)(struct wiphy *wiphy, struct net_device *dev, const u8 *addr);
  void (*rfkill_poll)(struct wiphy *wiphy);
#ifdef CONFIG_NL80211_TESTMODE;
  int (*testmode_cmd)(struct wiphy *wiphy, struct wireless_dev *wdev, void *data, int len);
  int (*testmode_dump)(struct wiphy *wiphy, struct sk_buff *skb,struct netlink_callback *cb, void *data,
#endif;
  int (*set_bitrate_mask)(struct wiphy *wiphy,struct net_device *dev,const u8 *peer, const struct cfg802
  int (*dump_survey)(struct wiphy *wiphy, struct net_device *netdev, int idx, struct survey_info *info);
  int (*set_pmksa)(struct wiphy *wiphy, struct net_device *netdev, struct cfg80211_pmksa *pmksa);
  int (*del_pmksa)(struct wiphy *wiphy, struct net_device *netdev, struct cfg80211_pmksa *pmksa);
  int (*flush_pmksa)(struct wiphy *wiphy, struct net_device *netdev);
  int (*remain_on_channel)(struct wiphy *wiphy,struct wireless_dev *wdev,struct ieee80211_channel *chan,
  int (*cancel_remain_on_channel)(struct wiphy *wiphy,struct wireless_dev *wdev, u64 cookie);
  int (*mgmt_tx)(struct wiphy *wiphy, struct wireless_dev *wdev,struct cfg80211_mgmt_tx_params *params, 
  int (*mgmt_tx_cancel_wait)(struct wiphy *wiphy,struct wireless_dev *wdev, u64 cookie);
  int (*set_power_mgmt)(struct wiphy *wiphy, struct net_device *dev, bool enabled, int timeout);
  int (*set_cqm_rssi_config)(struct wiphy *wiphy,struct net_device *dev, s32 rssi_thold, u32 rssi_hyst);
  int (*set_cqm_rssi_range_config)(struct wiphy *wiphy,struct net_device *dev, s32 rssi_low, s32 rssi_hi
  int (*set_cqm_txe_config)(struct wiphy *wiphy,struct net_device *dev, u32 rate, u32 pkts, u32 intvl);
  void (*mgmt_frame_register)(struct wiphy *wiphy,struct wireless_dev *wdev, u16 frame_type, bool reg);
  int (*set_antenna)(struct wiphy *wiphy, u32 tx_ant, u32 rx_ant);
  int (*get_antenna)(struct wiphy *wiphy, u32 *tx_ant, u32 *rx_ant);
  int (*sched_scan_start)(struct wiphy *wiphy,struct net_device *dev, struct cfg80211_sched_scan_request
  int (*sched_scan_stop)(struct wiphy *wiphy, struct net_device *dev, u64 reqid);
  int (*set_rekey_data)(struct wiphy *wiphy, struct net_device *dev, struct cfg80211_gtk_rekey_data *dat
  int (*tdls_mgmt)(struct wiphy *wiphy, struct net_device *dev,const u8 *peer, u8 action_code,  u8 dialo
  int (*tdls_oper)(struct wiphy *wiphy, struct net_device *dev, const u8 *peer, enum nl80211_tdls_operat
  int (*probe_client)(struct wiphy *wiphy, struct net_device *dev, const u8 *peer, u64 *cookie);
  int (*set_noack_map)(struct wiphy *wiphy,struct net_device *dev, u16 noack_map);
  int (*get_channel)(struct wiphy *wiphy,struct wireless_dev *wdev, struct cfg80211_chan_def *chandef);
  int (*start_p2p_device)(struct wiphy *wiphy, struct wireless_dev *wdev);
  void (*stop_p2p_device)(struct wiphy *wiphy, struct wireless_dev *wdev);
  int (*set_mac_acl)(struct wiphy *wiphy, struct net_device *dev, const struct cfg80211_acl_data *params
  int (*start_radar_detection)(struct wiphy *wiphy,struct net_device *dev,struct cfg80211_chan_def *char
  int (*update_ft_ies)(struct wiphy *wiphy, struct net_device *dev, struct cfg80211_update_ft_ies_params
  int (*crit_proto_start)(struct wiphy *wiphy,struct wireless_dev *wdev,enum nl80211_crit_proto_id proto
  void (*crit_proto_stop)(struct wiphy *wiphy, struct wireless_dev *wdev);
  int (*set_coalesce)(struct wiphy *wiphy, struct cfg80211_coalesce *coalesce);
  int (*channel_switch)(struct wiphy *wiphy,struct net_device *dev, struct cfg80211_csa_settings *params
  int (*set_qos_map)(struct wiphy *wiphy,struct net_device *dev, struct cfg80211_qos_map *qos_map);
  int (*set_ap_chanwidth)(struct wiphy *wiphy, struct net_device *dev, struct cfg80211_chan_def *chandef
  int (*add_tx_ts)(struct wiphy *wiphy, struct net_device *dev,u8 tsid, const u8 *peer, u8 user_prio, u1
  int (*del_tx_ts)(struct wiphy *wiphy, struct net_device *dev, u8 tsid, const u8 *peer);
  int (*tdls_channel_switch)(struct wiphy *wiphy,struct net_device *dev,const u8 *addr, u8 oper_class, s
  void (*tdls_cancel_channel_switch)(struct wiphy *wiphy,struct net_device *dev, const u8 *addr);
  int (*start_nan)(struct wiphy *wiphy, struct wireless_dev *wdev, struct cfg80211_nan_conf *conf);
  void (*stop_nan)(struct wiphy *wiphy, struct wireless_dev *wdev);
  int (*add_nan_func)(struct wiphy *wiphy, struct wireless_dev *wdev, struct cfg80211_nan_func *nan_func
  void (*del_nan_func)(struct wiphy *wiphy, struct wireless_dev *wdev, u64 cookie);
  int (*nan_change_conf)(struct wiphy *wiphy,struct wireless_dev *wdev,struct cfg80211_nan_conf *conf, u
  int (*set_multicast_to_unicast)(struct wiphy *wiphy,struct net_device *dev, const bool enabled);
  int (*set_pmk)(struct wiphy *wiphy, struct net_device *dev, const struct cfg80211_pmk_conf *conf);
  int (*del_pmk)(struct wiphy *wiphy, struct net_device *dev, const u8 *aa);
};
```

Members

`suspend` wiphy device needs to be suspended. The variable **wow** will be `NULL` or contain the enabled Wake-on-Wireless triggers that are configured for the device.

`resume` wiphy device needs to be resumed

`set_wakeup` Called when WoWLAN is enabled/disabled, use this callback to call `device_set_wakeup_enable()` to enable/disable wakeup from the device.

`add_virtual_intf` create a new virtual interface with the given name, must set the struct wireless_dev's iftype. Beware: You must create the new netdev in the wiphy's network namespace! Returns the struct wireless_dev, or an ERR_PTR. For P2P device wdevs, the driver must also set the address member in the wdev.

`del_virtual_intf` remove the virtual interface

`change_virtual_intf` change type/configuration of virtual interface, keep the struct wireless_dev's iftype updated.

`add_key` add a key with the given parameters. **mac_addr** will be `NULL` when adding a group key.

`get_key` get information about the key with the given parameters. **mac_addr** will be `NULL` when requesting information for a group key. All pointers given to the **callback** function need not be valid after it returns. This function should return an error if it is not possible to retrieve the key, -ENOENT if it doesn't exist.

`del_key` remove a key given the **mac_addr** (`NULL` for a group key) and **key_index**, return -ENOENT if the key doesn't exist.

`set_default_key` set the default key on an interface

`set_default_mgmt_key` set the default management frame key on an interface

`start_ap` Start acting in AP mode defined by the parameters.

`change_beacon` Change the beacon parameters for an access point mode interface. This should reject the call when AP mode wasn't started.

`stop_ap` Stop being an AP, including stopping beaconing.

add_station Add a new station.

del_station Remove a station

change_station Modify a given station. Note that flags changes are not much validated in cfg80211, in particular the auth/assoc/authorized flags might come to the driver in invalid combinations – make sure to check them, also against the existing state! Drivers must call `cfg80211_check_station_change()` to validate the information.

get_station get station information for the station identified by **mac**

dump_station dump station callback – resume dump at index **idx**

add_mpath add a fixed mesh path

del_mpath delete a given mesh path

change_mpath change a given mesh path

get_mpath get a mesh path for the given parameters

dump_mpath dump mesh path callback – resume dump at index **idx**

get_mpp get a mesh proxy path for the given parameters

dump_mpp dump mesh proxy path callback – resume dump at index **idx**

get_mesh_config Get the current mesh configuration

update_mesh_config Update mesh parameters on a running mesh. The mask is a bitfield which tells us which parameters to set, and which to leave alone.

join_mesh join the mesh network with the specified parameters (invoked with the wireless_dev mutex held)

leave_mesh leave the current mesh network (invoked with the wireless_dev mutex held)

join_ocb join the OCB network with the specified parameters (invoked with the wireless_dev mutex held)

leave_ocb leave the current OCB network (invoked with the wireless_dev mutex held)

change_bss Modify parameters for a given BSS.

set_txq_params Set TX queue parameters

libertas_set_mesh_channel Only for backward compatibility for libertas, as it doesn't implement join_mesh and needs to set the channel to join the mesh instead.

set_monitor_channel Set the monitor mode channel for the device. If other interfaces are active this callback should reject the configuration. If no interfaces are active or the device is down, the channel should be stored for when a monitor interface becomes active.

scan Request to do a scan. If returning zero, the scan request is given the driver, and will be valid until passed to *`cfg80211_scan_done()`*. For scan results, call *`cfg80211_inform_bss()`*; you can call this outside the scan/scan_done bracket too.

abort_scan Tell the driver to abort an ongoing scan. The driver shall indicate the status of the scan through *`cfg80211_scan_done()`*.

auth Request to authenticate with the specified peer (invoked with the wireless_dev mutex held)

assoc Request to (re)associate with the specified peer (invoked with the wireless_dev mutex held)

deauth Request to deauthenticate from the specified peer (invoked with the wireless_dev mutex held)

disassoc Request to disassociate from the specified peer (invoked with the wireless_dev mutex held)

connect Connect to the ESS with the specified parameters. When connected, call *`cfg80211_connect_result()`*/*`cfg80211_connect_bss()`* with status code `WLAN_STATUS_SUCCESS`. If the connection fails for some reason, call *`cfg80211_connect_result()`*/*`cfg80211_connect_bss()`* with the status code from the AP or *`cfg80211_connect_timeout()`* if no frame with status code was received. The driver is allowed to roam to other BSSes within the ESS when the other BSS matches the connect parameters. When such roaming is initiated by the driver, the driver is expected to verify that the target matches the configured security parameters and to use Reassociation Request frame instead of Association Request frame. The connect function can also be used to request the driver to perform a specific roam when connected to an ESS. In that case, the prev_bssid parameter is set to the BSSID of the currently associated BSS as an indication of requesting reassociation. In both the driver-initiated and new `connect()` call initiated roaming cases, the result of roaming is indicated with a call to *`cfg80211_roamed()`*. (invoked with the wireless_dev mutex held)

`update_connect_params` Update the connect parameters while connected to a BSS. The updated parameters can be used by driver/firmware for subsequent BSS selection (roaming) decisions and to form the Authentication/(Re)Association Request frames. This call does not request an immediate disassociation or reassociation with the current BSS, i.e., this impacts only subsequent (re)associations. The bits in changed are defined in `enum cfg80211_connect_params_changed`. (invoked with the wireless_dev mutex held)

`disconnect` Disconnect from the BSS/ESS or stop connection attempts if connection is in progress. Once done, call *`cfg80211_disconnected()`* in case connection was already established (invoked with the wireless_dev mutex held), otherwise call *`cfg80211_connect_timeout()`*.

`join_ibss` Join the specified IBSS (or create if necessary). Once done, call *`cfg80211_ibss_joined()`*, also call that function when changing BSSID due to a merge. (invoked with the wireless_dev mutex held)

`leave_ibss` Leave the IBSS. (invoked with the wireless_dev mutex held)

`set_mcast_rate` Set the specified multicast rate (only if vif is in ADHOC or MESH mode)

`set_wiphy_params` Notify that wiphy parameters have changed; **changed** bitfield (see *`enum wiphy_params_flags`*) describes which values have changed. The actual parameter values are available in struct wiphy. If returning an error, no value should be changed.

`set_tx_power` set the transmit power according to the parameters, the power passed is in mBm, to get dBm use `MBM_TO_DBM()`. The wdev may be `NULL` if power was set for the wiphy, and will always be `NULL` unless the driver supports per-vif TX power (as advertised by the nl80211 feature flag.)

`get_tx_power` store the current TX power into the dbm variable; return 0 if successful

`set_wds_peer` set the WDS peer for a WDS interface

`rfkill_poll` polls the hw rfkill line, use cfg80211 reporting functions to adjust rfkill hw state

`testmode_cmd` run a test mode command; **wdev** may be `NULL`

`testmode_dump` Implement a test mode dump. The cb->args[2] and up may be used by the function, but 0 and 1 must not be touched. Additionally, return error codes other than -ENOBUFS and -ENOENT will terminate the dump and return to userspace with an error, so be careful. If any data was passed in from userspace then the data/len arguments will be present and point to the data contained in `NL80211_ATTR_TESTDATA`.

`set_bitrate_mask` set the bitrate mask configuration

`dump_survey` get site survey information.

`set_pmksa` Cache a PMKID for a BSSID. This is mostly useful for fullmac devices running firmwares capable of generating the (re) association RSN IE. It allows for faster roaming between WPA2 BSSIDs.

`del_pmksa` Delete a cached PMKID.

`flush_pmksa` Flush all cached PMKIDs.

`remain_on_channel` Request the driver to remain awake on the specified channel for the specified duration to complete an off-channel operation (e.g., public action frame exchange). When the driver is ready on the requested channel, it must indicate this with an event notification by calling *`cfg80211_ready_on_channel()`*.

`cancel_remain_on_channel` Cancel an on-going remain-on-channel operation. This allows the operation to be terminated prior to timeout based on the duration value.

mgmt_tx Transmit a management frame.

mgmt_tx_cancel_wait Cancel the wait time from transmitting a management frame on another channel

set_power_mgmt Configure WLAN power management. A timeout value of -1 allows the driver to adjust the dynamic ps timeout value.

set_cqm_rssi_config Configure connection quality monitor RSSI threshold. After configuration, the driver should (soon) send an event indicating the current level is above/below the configured threshold; this may need some care when the configuration is changed (without first being disabled.)

set_cqm_rssi_range_config Configure two RSSI thresholds in the connection quality monitor. An event is to be sent only when the signal level is found to be outside the two values. The driver should set `NL80211_EXT_FEATURE_CQM_RSSI_LIST` if this method is implemented. If it is provided then there's no point providing **set_cqm_rssi_config**.

set_cqm_txe_config Configure connection quality monitor TX error thresholds.

mgmt_frame_register Notify driver that a management frame type was registered. The callback is allowed to sleep.

set_antenna Set antenna configuration (tx_ant, rx_ant) on the device. Parameters are bitmaps of allowed antennas to use for TX/RX. Drivers may reject TX/RX mask combinations they cannot support by returning -EINVAL (also see nl80211.h **NL80211_ATTR_WIPHY_ANTENNA_TX**).

get_antenna Get current antenna configuration from device (tx_ant, rx_ant).

sched_scan_start Tell the driver to start a scheduled scan.

sched_scan_stop Tell the driver to stop an ongoing scheduled scan with given request id. This call must stop the scheduled scan and be ready for starting a new one before it returns, i.e. **sched_scan_start** may be called immediately after that again and should not fail in that case. The driver should not call `cfg80211_sched_scan_stopped()` for a requested stop (when this method returns 0).

set_rekey_data give the data necessary for GTK rekeying to the driver

tdls_mgmt Transmit a TDLS management frame.

tdls_oper Perform a high-level TDLS operation (e.g. TDLS link setup).

probe_client probe an associated client, must return a cookie that it later passes to `cfg80211_probe_status()`.

set_noack_map Set the NoAck Map for the TIDs.

get_channel Get the current operating channel for the virtual interface. For monitor interfaces, it should return `NULL` unless there's a single current monitoring channel.

start_p2p_device Start the given P2P device.

stop_p2p_device Stop the given P2P device.

set_mac_acl Sets MAC address control list in AP and P2P GO mode. Parameters include ACL policy, an array of MAC address of stations and the number of MAC addresses. If there is already a list in driver this new list replaces the existing one. Driver has to clear its ACL when number of MAC addresses entries is passed as 0. Drivers which advertise the support for MAC based ACL have to implement this callback.

start_radar_detection Start radar detection in the driver.

update_ft_ies Provide updated Fast BSS Transition information to the driver. If the SME is in the driver/firmware, this information can be used in building Authentication and Reassociation Request frames.

crit_proto_start Indicates a critical protocol needs more link reliability for a given duration (milliseconds). The protocol is provided so the driver can take the most appropriate actions.

crit_proto_stop Indicates critical protocol no longer needs increased link reliability. This operation can not fail.

set_coalesce Set coalesce parameters.

channel_switch initiate channel-switch procedure (with CSA). Driver is responsible for veryfing if the switch is possible. Since this is inherently tricky driver may decide to disconnect an interface later with `cfg80211_stop_iface()`. This doesn't mean driver can accept everything. It should do it's best to verify requests and reject them as soon as possible.

set_qos_map Set QoS mapping information to the driver

set_ap_chanwidth Set the AP (including P2P GO) mode channel width for the given interface This is used e.g. for dynamic HT 20/40 MHz channel width changes during the lifetime of the BSS.

add_tx_ts validate (if admitted_time is 0) or add a TX TS to the device with the given parameters; action frame exchange has been handled by userspace so this just has to modify the TX path to take the TS into account. If the admitted time is 0 just validate the parameters to make sure the session can be created at all; it is valid to just always return success for that but that may result in inefficient behaviour (handshake with the peer followed by immediate teardown when the addition is later rejected)

del_tx_ts remove an existing TX TS

tdls_channel_switch Start channel-switching with a TDLS peer. The driver is responsible for continually initiating channel-switching operations and returning to the base channel for communication with the AP.

tdls_cancel_channel_switch Stop channel-switching with a TDLS peer. Both peers must be on the base channel when the call completes.

start_nan Start the NAN interface.

stop_nan Stop the NAN interface.

add_nan_func Add a NAN function. Returns negative value on failure. On success **nan_func** ownership is transferred to the driver and it may access it outside of the scope of this function. The driver should free the **nan_func** when no longer needed by calling `cfg80211_free_nan_func()`. On success the driver should assign an instance_id in the provided **nan_func**.

del_nan_func Delete a NAN function.

nan_change_conf changes NAN configuration. The changed parameters must be specified in **changes** (using `enum cfg80211_nan_conf_changes`); All other parameters must be ignored.

set_multicast_to_unicast configure multicast to unicast conversion for BSS

set_pmk configure the PMK to be used for offloaded 802.1X 4-Way handshake. If not deleted through **del_pmk** the PMK remains valid until disconnect upon which the driver should clear it. (invoked with the wireless_dev mutex held)

del_pmk delete the previously configured PMK for the given authenticator. (invoked with the wireless_dev mutex held)

Description

This struct is registered by fullmac card drivers and/or wireless stacks in order to handle configuration requests on their interfaces.

All callbacks except where otherwise noted should return 0 on success or a negative error code.

All operations are currently invoked under rtnl for consistency with the wireless extensions but this is subject to reevaluation as soon as this code is used more widely and we have a first user without wext.

struct **vif_params**
describes virtual interface parameters

Definition

```
struct vif_params {
  u32 flags;
  int use_4addr;
  u8 macaddr[ETH_ALEN];
  const u8 *vht_mumimo_groups;
  const u8 *vht_mumimo_follow_addr;
};
```

Members

flags monitor interface flags, unchanged if 0, otherwise `MONITOR_FLAG_CHANGED` will be set

use_4addr use 4-address frames

macaddr address to use for this virtual interface. If this parameter is set to zero address the driver may determine the address as needed. This feature is only fully supported by drivers that enable the `NL80211_FEATURE_MAC_ON_CREATE` flag. Others may support creating * only p2p devices with specified MAC.

vht_mumimo_groups MU-MIMO groupID, used for monitoring MU-MIMO packets belonging to that MU-MIMO groupID; `NULL` if not changed

vht_mumimo_follow_addr MU-MIMO follow address, used for monitoring MU-MIMO packets going to the specified station; `NULL` if not changed

struct **key_params**
key information

Definition

```
struct key_params {
  const u8 *key;
  const u8 *seq;
  int key_len;
  int seq_len;
  u32 cipher;
};
```

Members

key key material

seq sequence counter (IV/PN) for TKIP and CCMP keys, only used with the `get_key()` callback, must be in little endian, length given by **seq_len**.

key_len length of key material

seq_len length of **seq**.

cipher cipher suite selector

Description

Information about a key

enum **survey_info_flags**
survey information flags

Constants

SURVEY_INFO_NOISE_DBM noise (in dBm) was filled in

SURVEY_INFO_IN_USE channel is currently being used

SURVEY_INFO_TIME active time (in ms) was filled in

SURVEY_INFO_TIME_BUSY busy time was filled in

SURVEY_INFO_TIME_EXT_BUSY extension channel busy time was filled in

SURVEY_INFO_TIME_RX receive time was filled in

SURVEY_INFO_TIME_TX transmit time was filled in

SURVEY_INFO_TIME_SCAN scan time was filled in

Description

Used by the driver to indicate which info in *struct survey_info* it has filled in during the `get_survey()`.

struct **survey_info**
channel survey response

Definition

```
struct survey_info {
  struct ieee80211_channel *channel;
  u64 time;
  u64 time_busy;
  u64 time_ext_busy;
  u64 time_rx;
  u64 time_tx;
  u64 time_scan;
  u32 filled;
  s8 noise;
};
```

Members

channel the channel this survey record reports, may be `NULL` for a single record to report global statistics

time amount of time in ms the radio was turn on (on the channel)

time_busy amount of time the primary channel was sensed busy

time_ext_busy amount of time the extension channel was sensed busy

time_rx amount of time the radio spent receiving data

time_tx amount of time the radio spent transmitting data

time_scan amount of time the radio spent for scanning

filled bitflag of flags from *enum survey_info_flags*

noise channel noise in dBm. This and all following fields are optional

Description

Used by `dump_survey()` to report back per-channel survey information.

This structure can later be expanded with things like channel duty cycle etc.

struct **cfg80211_beacon_data**
beacon data

Definition

```
struct cfg80211_beacon_data {
  const u8 *head, *tail;
  const u8 *beacon_ies;
  const u8 *proberesp_ies;
  const u8 *assocresp_ies;
  const u8 *probe_resp;
  size_t head_len, tail_len;
  size_t beacon_ies_len;
  size_t proberesp_ies_len;
  size_t assocresp_ies_len;
  size_t probe_resp_len;
};
```

Members

head head portion of beacon (before TIM IE) or NULL if not changed

tail tail portion of beacon (after TIM IE) or NULL if not changed

beacon_ies extra information element(s) to add into Beacon frames or NULL

proberesp_ies extra information element(s) to add into Probe Response frames or NULL

assocresp_ies extra information element(s) to add into (Re)Association Response frames or NULL

probe_resp probe response template (AP mode only)

head_len length of **head**

tail_len length of **tail**

beacon_ies_len length of beacon_ies in octets

proberesp_ies_len length of proberesp_ies in octets

assocresp_ies_len length of assocresp_ies in octets

probe_resp_len length of probe response template (**probe_resp**)

struct **cfg80211_ap_settings**
AP configuration

Definition

```
struct cfg80211_ap_settings {
  struct cfg80211_chan_def chandef;
  struct cfg80211_beacon_data beacon;
  int beacon_interval, dtim_period;
  const u8 *ssid;
  size_t ssid_len;
  enum nl80211_hidden_ssid hidden_ssid;
  struct cfg80211_crypto_settings crypto;
  bool privacy;
  enum nl80211_auth_type auth_type;
  enum nl80211_smps_mode smps_mode;
  int inactivity_timeout;
  u8 p2p_ctwindow;
  bool p2p_opp_ps;
  const struct cfg80211_acl_data *acl;
  bool pbss;
  struct cfg80211_bitrate_mask beacon_rate;
  const struct ieee80211_ht_cap *ht_cap;
  const struct ieee80211_vht_cap *vht_cap;
  bool ht_required, vht_required;
};
```

Members

chandef defines the channel to use

beacon beacon data

beacon_interval beacon interval

dtim_period DTIM period

ssid SSID to be used in the BSS (note: may be NULL if not provided from user space)

ssid_len length of **ssid**

hidden_ssid whether to hide the SSID in Beacon/Probe Response frames

crypto crypto settings

privacy the BSS uses privacy

auth_type Authentication type (algorithm)

smps_mode SMPS mode

inactivity_timeout time in seconds to determine station's inactivity.

p2p_ctwindow P2P CT Window

p2p_opp_ps P2P opportunistic PS

acl ACL configuration used by the drivers which has support for MAC address based access control

pbss If set, start as a PCP instead of AP. Relevant for DMG networks.

beacon_rate bitrate to be used for beacons

ht_cap HT capabilities (or NULL if HT isn't enabled)

vht_cap VHT capabilities (or NULL if VHT isn't enabled)

ht_required stations must support HT

vht_required stations must support VHT

Description

Used to configure an AP interface.

struct **station_parameters**
station parameters

Definition

```
struct station_parameters {
  const u8 *supported_rates;
  struct net_device *vlan;
  u32 sta_flags_mask, sta_flags_set;
  u32 sta_modify_mask;
  int listen_interval;
  u16 aid;
  u16 peer_aid;
  u8 supported_rates_len;
  u8 plink_action;
  u8 plink_state;
  const struct ieee80211_ht_cap *ht_capa;
  const struct ieee80211_vht_cap *vht_capa;
  u8 uapsd_queues;
  u8 max_sp;
  enum nl80211_mesh_power_mode local_pm;
  u16 capability;
  const u8 *ext_capab;
  u8 ext_capab_len;
  const u8 *supported_channels;
  u8 supported_channels_len;
  const u8 *supported_oper_classes;
  u8 supported_oper_classes_len;
  u8 opmode_notif;
  bool opmode_notif_used;
  int support_p2p_ps;
};
```

Members

supported_rates supported rates in IEEE 802.11 format (or NULL for no change)

vlan vlan interface station should belong to

sta_flags_mask station flags that changed (bitmask of BIT(NL80211_STA_FLAG_...))

sta_flags_set station flags values (bitmask of BIT(NL80211_STA_FLAG_...))

sta_modify_mask bitmap indicating which parameters changed (for those that don't have a natural "no change" value), see enum `station_parameters_apply_mask`

listen_interval listen interval or -1 for no change

aid AID or zero for no change

peer_aid mesh peer AID or zero for no change

supported_rates_len number of supported rates

plink_action plink action to take

plink_state set the peer link state for a station

ht_capa HT capabilities of station

vht_capa VHT capabilities of station

uapsd_queues bitmap of queues configured for uapsd. same format as the AC bitmap in the QoS info field

max_sp max Service Period. same format as the MAX_SP in the QoS info field (but already shifted down)

local_pm local link-specific mesh power save mode (no change when set to unknown)

capability station capability

ext_capab extended capabilities of the station

ext_capab_len number of extended capabilities

supported_channels supported channels in IEEE 802.11 format

supported_channels_len number of supported channels

supported_oper_classes supported oper classes in IEEE 802.11 format

supported_oper_classes_len number of supported operating classes

opmode_notif operating mode field from Operating Mode Notification

opmode_notif_used information if operating mode field is used

support_p2p_ps information if station supports P2P PS mechanism

Description

Used to change and create a new station.

enum **rate_info_flags**
bitrate info flags

Constants

RATE_INFO_FLAGS_MCS mcs field filled with HT MCS

RATE_INFO_FLAGS_VHT_MCS mcs field filled with VHT MCS

RATE_INFO_FLAGS_SHORT_GI 400ns guard interval

RATE_INFO_FLAGS_60G 60GHz MCS

Description

Used by the driver to indicate the specific rate transmission type for 802.11n transmissions.

struct **rate_info**
bitrate information

Definition

```
struct rate_info {
  u8 flags;
  u8 mcs;
  u16 legacy;
  u8 nss;
  u8 bw;
};
```

Members

flags bitflag of flags from *enum rate_info_flags*

mcs mcs index if struct describes a 802.11n bitrate

legacy bitrate in 100kbit/s for 802.11abg

nss number of streams (VHT only)

bw bandwidth (from enum `rate_info_bw`)

Description

Information about a receiving or transmitting bitrate

struct **station_info**
station information

Definition

```
struct station_info {
  u64 filled;
  u32 connected_time;
  u32 inactive_time;
  u64 rx_bytes;
  u64 tx_bytes;
  u16 llid;
  u16 plid;
  u8 plink_state;
  s8 signal;
  s8 signal_avg;
  u8 chains;
  s8 chain_signal[IEEE80211_MAX_CHAINS];
  s8 chain_signal_avg[IEEE80211_MAX_CHAINS];
  struct rate_info txrate;
  struct rate_info rxrate;
  u32 rx_packets;
  u32 tx_packets;
  u32 tx_retries;
  u32 tx_failed;
  u32 rx_dropped_misc;
  struct sta_bss_parameters bss_param;
  struct nl80211_sta_flag_update sta_flags;
  int generation;
  const u8 *assoc_req_ies;
  size_t assoc_req_ies_len;
  u32 beacon_loss_count;
  s64 t_offset;
  enum nl80211_mesh_power_mode local_pm;
  enum nl80211_mesh_power_mode peer_pm;
  enum nl80211_mesh_power_mode nonpeer_pm;
  u32 expected_throughput;
  u64 rx_beacon;
  u64 rx_duration;
  u8 rx_beacon_signal_avg;
  struct cfg80211_tid_stats pertid[IEEE80211_NUM_TIDS + 1];
};
```

Members

filled bitflag of flags using the bits of enum `nl80211_sta_info` to indicate the relevant values in this struct for them

connected_time time(in secs) since a station is last connected

inactive_time time since last station activity (tx/rx) in milliseconds

rx_bytes bytes (size of MPDUs) received from this station

tx_bytes bytes (size of MPDUs) transmitted to this station

llid mesh local link id

plid mesh peer link id

plink_state mesh peer link state

signal The signal strength, type depends on the wiphy's signal_type. For CFG80211_SIGNAL_TYPE_MBM, value is expressed in _dBm_.

signal_avg Average signal strength, type depends on the wiphy's signal_type. For CFG80211_SIGNAL_TYPE_MBM, value is expressed in _dBm_.

chains bitmask for filled values in **chain_signal**, **chain_signal_avg**

chain_signal per-chain signal strength of last received packet in dBm

chain_signal_avg per-chain signal strength average in dBm

txrate current unicast bitrate from this station

rxrate current unicast bitrate to this station

rx_packets packets (MSDUs & MMPDUs) received from this station

tx_packets packets (MSDUs & MMPDUs) transmitted to this station

tx_retries cumulative retry counts (MPDUs)

tx_failed number of failed transmissions (MPDUs) (retries exceeded, no ACK)

rx_dropped_misc Dropped for un-specified reason.

bss_param current BSS parameters

sta_flags station flags mask & values

generation generation number for nl80211 dumps. This number should increase every time the list of stations changes, i.e. when a station is added or removed, so that userspace can tell whether it got a consistent snapshot.

assoc_req_ies IEs from (Re)Association Request. This is used only when in AP mode with drivers that do not use user space MLME/SME implementation. The information is provided for the *cfg80211_new_sta()* calls to notify user space of the IEs.

assoc_req_ies_len Length of assoc_req_ies buffer in octets.

beacon_loss_count Number of times beacon loss event has triggered.

t_offset Time offset of the station relative to this host.

local_pm local mesh STA power save mode

peer_pm peer mesh STA power save mode

nonpeer_pm non-peer mesh STA power save mode

expected_throughput expected throughput in kbps (including 802.11 headers) towards this station.

rx_beacon number of beacons received from this peer

rx_duration aggregate PPDU duration(usecs) for all the frames from a peer

rx_beacon_signal_avg signal strength average (in dBm) for beacons received from this peer

pertid per-TID statistics, see `struct cfg80211_tid_stats`, using the last (IEEE80211_NUM_TIDS) index for MSDUs not encapsulated in QoS-MPDUs.

Description

Station information filled by driver for `get_station()` and dump_station.

enum **`monitor_flags`**
monitor flags

Constants

`MONITOR_FLAG_CHANGED` set if the flags were changed

`MONITOR_FLAG_FCSFAIL` pass frames with bad FCS

`MONITOR_FLAG_PLCPFAIL` pass frames with bad PLCP

`MONITOR_FLAG_CONTROL` pass control frames

`MONITOR_FLAG_OTHER_BSS` disable BSSID filtering

`MONITOR_FLAG_COOK_FRAMES` report frames after processing

`MONITOR_FLAG_ACTIVE` active monitor, ACKs frames on its MAC address

Description

Monitor interface configuration flags. Note that these must be the bits according to the nl80211 flags.

enum **`mpath_info_flags`**
mesh path information flags

Constants

`MPATH_INFO_FRAME_QLEN` **frame_qlen** filled

`MPATH_INFO_SN` **sn** filled

`MPATH_INFO_METRIC` **metric** filled

`MPATH_INFO_EXPTIME` **exptime** filled

`MPATH_INFO_DISCOVERY_TIMEOUT` **discovery_timeout** filled

`MPATH_INFO_DISCOVERY_RETRIES` **discovery_retries** filled

`MPATH_INFO_FLAGS` **flags** filled

Description

Used by the driver to indicate which info in *`struct mpath_info`* it has filled in during `get_station()` or `dump_station()`.

struct **`mpath_info`**
mesh path information

Definition

```
struct mpath_info {
  u32 filled;
  u32 frame_qlen;
  u32 sn;
  u32 metric;
  u32 exptime;
  u32 discovery_timeout;
  u8 discovery_retries;
  u8 flags;
  int generation;
};
```

Members

`filled` bitfield of flags from *`enum mpath_info_flags`*

`frame_qlen` number of queued frames for this destination

sn target sequence number

metric metric (cost) of this mesh path

exptime expiration time for the mesh path from now, in msecs

discovery_timeout total mesh path discovery timeout, in msecs

discovery_retries mesh path discovery retries

flags mesh path flags

generation generation number for nl80211 dumps. This number should increase every time the list of mesh paths changes, i.e. when a station is added or removed, so that userspace can tell whether it got a consistent snapshot.

Description

Mesh path information filled by driver for `get_mpath()` and `dump_mpath()`.

struct **bss_parameters**
BSS parameters

Definition

```
struct bss_parameters {
  int use_cts_prot;
  int use_short_preamble;
  int use_short_slot_time;
  const u8 *basic_rates;
  u8 basic_rates_len;
  int ap_isolate;
  int ht_opmode;
  s8 p2p_ctwindow, p2p_opp_ps;
};
```

Members

use_cts_prot Whether to use CTS protection (0 = no, 1 = yes, -1 = do not change)

use_short_preamble Whether the use of short preambles is allowed (0 = no, 1 = yes, -1 = do not change)

use_short_slot_time Whether the use of short slot time is allowed (0 = no, 1 = yes, -1 = do not change)

basic_rates basic rates in IEEE 802.11 format (or NULL for no change)

basic_rates_len number of basic rates

ap_isolate do not forward packets between connected stations

ht_opmode HT Operation mode (u16 = opmode, -1 = do not change)

p2p_ctwindow P2P CT Window (-1 = no change)

p2p_opp_ps P2P opportunistic PS (-1 = no change)

Description

Used to change BSS parameters (mainly for AP mode).

struct **ieee80211_txq_params**
TX queue parameters

Definition

```
struct ieee80211_txq_params {
  enum nl80211_ac ac;
  u16 txop;
  u16 cwmin;
  u16 cwmax;
  u8 aifs;
};
```

Members

ac AC identifier

txop Maximum burst time in units of 32 usecs, 0 meaning disabled

cwmin Minimum contention window [a value of the form 2^n-1 in the range 1..32767]

cwmax Maximum contention window [a value of the form 2^n-1 in the range 1..32767]

aifs Arbitration interframe space [0..255]

struct **cfg80211_crypto_settings**
Crypto settings

Definition

```
struct cfg80211_crypto_settings {
  u32 wpa_versions;
  u32 cipher_group;
  int n_ciphers_pairwise;
  u32 ciphers_pairwise[NL80211_MAX_NR_CIPHER_SUITES];
  int n_akm_suites;
  u32 akm_suites[NL80211_MAX_NR_AKM_SUITES];
  bool control_port;
  __be16 control_port_ethertype;
  bool control_port_no_encrypt;
  struct key_params *wep_keys;
  int wep_tx_key;
  const u8 *psk;
};
```

Members

wpa_versions indicates which, if any, WPA versions are enabled (from enum nl80211_wpa_versions)

cipher_group group key cipher suite (or 0 if unset)

n_ciphers_pairwise number of AP supported unicast ciphers

ciphers_pairwise unicast key cipher suites

n_akm_suites number of AKM suites

akm_suites AKM suites

control_port Whether user space controls IEEE 802.1X port, i.e., sets/clears NL80211_STA_FLAG_AUTHORIZED. If true, the driver is required to assume that the port is unauthorized until authorized by user space. Otherwise, port is marked authorized by default.

control_port_ethertype the control port protocol that should be allowed through even on unauthorized ports

control_port_no_encrypt TRUE to prevent encryption of control port protocol frames.

wep_keys static WEP keys, if not NULL points to an array of CFG80211_MAX_WEP_KEYS WEP keys

wep_tx_key key index (0..3) of the default TX static WEP key

psk PSK (for devices supporting 4-way-handshake offload)

struct **cfg80211_auth_request**
Authentication request data

Definition

```
struct cfg80211_auth_request {
  struct cfg80211_bss *bss;
  const u8 *ie;
  size_t ie_len;
  enum nl80211_auth_type auth_type;
  const u8 *key;
  u8 key_len, key_idx;
  const u8 *auth_data;
  size_t auth_data_len;
};
```

Members

bss The BSS to authenticate with, the callee must obtain a reference to it if it needs to keep it.

ie Extra IEs to add to Authentication frame or NULL

ie_len Length of ie buffer in octets

auth_type Authentication type (algorithm)

key WEP key for shared key authentication

key_len length of WEP key for shared key authentication

key_idx index of WEP key for shared key authentication

auth_data Fields and elements in Authentication frames. This contains the authentication frame body (non-IE and IE data), excluding the Authentication algorithm number, i.e., starting at the Authentication transaction sequence number field.

auth_data_len Length of auth_data buffer in octets

Description

This structure provides information needed to complete IEEE 802.11 authentication.

struct **cfg80211_assoc_request**
(Re)Association request data

Definition

```
struct cfg80211_assoc_request {
  struct cfg80211_bss *bss;
  const u8 *ie, *prev_bssid;
  size_t ie_len;
  struct cfg80211_crypto_settings crypto;
  bool use_mfp;
  u32 flags;
  struct ieee80211_ht_cap ht_capa;
  struct ieee80211_ht_cap ht_capa_mask;
  struct ieee80211_vht_cap vht_capa, vht_capa_mask;
  const u8 *fils_kek;
  size_t fils_kek_len;
  const u8 *fils_nonces;
};
```

Members

bss The BSS to associate with. If the call is successful the driver is given a reference that it must give back to `cfg80211_send_rx_assoc()` or to *`cfg80211_assoc_timeout()`*. To ensure proper refcounting, new association requests while already associating must be rejected.

ie Extra IEs to add to (Re)Association Request frame or NULL

prev_bssid previous BSSID, if not NULL use reassociate frame. This is used to indicate a request to reassociate within the ESS instead of a request do the initial association with the ESS. When included, this is set to the BSSID of the current association, i.e., to the value that is included in the Current AP address field of the Reassociation Request frame.

ie_len Length of ie buffer in octets

crypto crypto settings

use_mfp Use management frame protection (IEEE 802.11w) in this association

flags See enum `cfg80211_assoc_req_flags`

ht_capa HT Capabilities over-rides. Values set in ht_capa_mask will be used in ht_capa. Un-supported values will be ignored.

ht_capa_mask The bits of ht_capa which are to be used.

vht_capa VHT capability override

vht_capa_mask VHT capability mask indicating which fields to use

fils_kek FILS KEK for protecting (Re)Association Request/Response frame or NULL if FILS is not used.

fils_kek_len Length of fils_kek in octets

fils_nonces FILS nonces (part of AAD) for protecting (Re)Association Request/Response frame or NULL if FILS is not used. This field starts with 16 octets of STA Nonce followed by 16 octets of AP Nonce.

Description

This structure provides information needed to complete IEEE 802.11 (re)association.

struct **cfg80211_deauth_request**
Deauthentication request data

Definition

```
struct cfg80211_deauth_request {
  const u8 *bssid;
  const u8 *ie;
  size_t ie_len;
  u16 reason_code;
  bool local_state_change;
};
```

Members

bssid the BSSID of the BSS to deauthenticate from

ie Extra IEs to add to Deauthentication frame or NULL

ie_len Length of ie buffer in octets

reason_code The reason code for the deauthentication

local_state_change if set, change local state only and do not set a deauth frame

Description

This structure provides information needed to complete IEEE 802.11 deauthentication.

struct **cfg80211_disassoc_request**
Disassociation request data

Definition

```
struct cfg80211_disassoc_request {
  struct cfg80211_bss *bss;
  const u8 *ie;
  size_t ie_len;
  u16 reason_code;
  bool local_state_change;
};
```

Members

bss the BSS to disassociate from

ie Extra IEs to add to Disassociation frame or NULL

ie_len Length of ie buffer in octets

reason_code The reason code for the disassociation

local_state_change This is a request for a local state only, i.e., no Disassociation frame is to be transmitted.

Description

This structure provides information needed to complete IEEE 802.11 disassociation.

struct **cfg80211_ibss_params**
IBSS parameters

Definition

```
struct cfg80211_ibss_params {
  const u8 *ssid;
  const u8 *bssid;
  struct cfg80211_chan_def chandef;
  const u8 *ie;
  u8 ssid_len, ie_len;
  u16 beacon_interval;
  u32 basic_rates;
  bool channel_fixed;
  bool privacy;
  bool control_port;
  bool userspace_handles_dfs;
  int mcast_rate[NUM_NL80211_BANDS];
  struct ieee80211_ht_cap ht_capa;
  struct ieee80211_ht_cap ht_capa_mask;
  struct key_params *wep_keys;
  int wep_tx_key;
};
```

Members

ssid The SSID, will always be non-null.

bssid Fixed BSSID requested, maybe be NULL, if set do not search for IBSSs with a different BSSID.

chandef defines the channel to use if no other IBSS to join can be found

ie information element(s) to include in the beacon

ssid_len The length of the SSID, will always be non-zero.

ie_len length of that

beacon_interval beacon interval to use

basic_rates bitmap of basic rates to use when creating the IBSS

channel_fixed The channel should be fixed – do not search for IBSSs to join on other channels.

privacy this is a protected network, keys will be configured after joining

control_port whether user space controls IEEE 802.1X port, i.e., sets/clears NL80211_STA_FLAG_AUTHORIZED. If true, the driver is required to assume that the port is unauthorized until authorized by user space. Otherwise, port is marked authorized by default.

userspace_handles_dfs whether user space controls DFS operation, i.e. changes the channel when a radar is detected. This is required to operate on DFS channels.

mcast_rate per-band multicast rate index + 1 (0: disabled)

ht_capa HT Capabilities over-rides. Values set in ht_capa_mask will be used in ht_capa. Un-supported values will be ignored.

ht_capa_mask The bits of ht_capa which are to be used.

wep_keys static WEP keys, if not NULL points to an array of CFG80211_MAX_WEP_KEYS WEP keys

wep_tx_key key index (0..3) of the default TX static WEP key

Description

This structure defines the IBSS parameters for the `join_ibss()` method.

struct **cfg80211_connect_params**
Connection parameters

Definition

```
struct cfg80211_connect_params {
  struct ieee80211_channel *channel;
  struct ieee80211_channel *channel_hint;
  const u8 *bssid;
  const u8 *bssid_hint;
  const u8 *ssid;
  size_t ssid_len;
  enum nl80211_auth_type auth_type;
  const u8 *ie;
  size_t ie_len;
  bool privacy;
  enum nl80211_mfp mfp;
  struct cfg80211_crypto_settings crypto;
  const u8 *key;
  u8 key_len, key_idx;
  u32 flags;
  int bg_scan_period;
  struct ieee80211_ht_cap ht_capa;
  struct ieee80211_ht_cap ht_capa_mask;
  struct ieee80211_vht_cap vht_capa;
  struct ieee80211_vht_cap vht_capa_mask;
  bool pbss;
  struct cfg80211_bss_selection bss_select;
  const u8 *prev_bssid;
  const u8 *fils_erp_username;
  size_t fils_erp_username_len;
  const u8 *fils_erp_realm;
  size_t fils_erp_realm_len;
  u16 fils_erp_next_seq_num;
  const u8 *fils_erp_rrk;
  size_t fils_erp_rrk_len;
  bool want_1x;
};
```

Members

channel The channel to use or NULL if not specified (auto-select based on scan results)

channel_hint The channel of the recommended BSS for initial connection or NULL if not specified

bssid The AP BSSID or NULL if not specified (auto-select based on scan results)

bssid_hint The recommended AP BSSID for initial connection to the BSS or NULL if not specified. Unlike the **bssid** parameter, the driver is allowed to ignore this **bssid_hint** if it has knowledge of a better BSS to use.

ssid SSID

ssid_len Length of ssid in octets

auth_type Authentication type (algorithm)

ie IEs for association request

ie_len Length of assoc_ie in octets

privacy indicates whether privacy-enabled APs should be used

mfp indicate whether management frame protection is used

crypto crypto settings

key WEP key for shared key authentication

key_len length of WEP key for shared key authentication

key_idx index of WEP key for shared key authentication

flags See enum cfg80211_assoc_req_flags

bg_scan_period Background scan period in seconds or -1 to indicate that default value is to be used.

ht_capa HT Capabilities over-rides. Values set in ht_capa_mask will be used in ht_capa. Un-supported values will be ignored.

ht_capa_mask The bits of ht_capa which are to be used.

vht_capa VHT Capability overrides

vht_capa_mask The bits of vht_capa which are to be used.

pbss if set, connect to a PCP instead of AP. Valid for DMG networks.

bss_select criteria to be used for BSS selection.

prev_bssid previous BSSID, if not NULL use reassociate frame. This is used to indicate a request to reassociate within the ESS instead of a request do the initial association with the ESS. When included, this is set to the BSSID of the current association, i.e., to the value that is included in the Current AP address field of the Reassociation Request frame.

fils_erp_username EAP re-authentication protocol (ERP) username part of the NAI or NULL if not specified. This is used to construct FILS wrapped data IE.

fils_erp_username_len Length of **fils_erp_username** in octets.

fils_erp_realm EAP re-authentication protocol (ERP) realm part of NAI or NULL if not specified. This specifies the domain name of ER server and is used to construct FILS wrapped data IE.

fils_erp_realm_len Length of **fils_erp_realm** in octets.

fils_erp_next_seq_num The next sequence number to use in the FILS ERP messages. This is also used to construct FILS wrapped data IE.

fils_erp_rrk ERP re-authentication Root Key (rRK) used to derive additional keys in FILS or NULL if not specified.

fils_erp_rrk_len Length of **fils_erp_rrk** in octets.

want_1x indicates user-space supports and wants to use 802.1X driver offload of 4-way handshake.

Description

This structure provides information needed to complete IEEE 802.11 authentication and association.

struct **cfg80211_pmksa**
PMK Security Association

Definition

```
struct cfg80211_pmksa {
  const u8 *bssid;
  const u8 *pmkid;
  const u8 *pmk;
  size_t pmk_len;
  const u8 *ssid;
  size_t ssid_len;
  const u8 *cache_id;
};
```

Members

bssid The AP's BSSID (may be NULL).

pmkid The identifier to refer a PMKSA.

pmk The PMK for the PMKSA identified by **pmkid**. This is used for key derivation by a FILS STA. Otherwise, NULL.

pmk_len Length of the **pmk**. The length of **pmk** can differ depending on the hash algorithm used to generate this.

ssid SSID to specify the ESS within which a PMKSA is valid when using FILS cache identifier (may be NULL).

ssid_len Length of the **ssid** in octets.

cache_id 2-octet cache identifier advertized by a FILS AP identifying the scope of PMKSA. This is valid only if **ssid_len** is non-zero (may be NULL).

Description

This structure is passed to the set/del_pmksa() method for PMKSA caching.

void **cfg80211_rx_mlme_mgmt**(struct net_device * *dev*, const u8 * *buf*, size_t *len*)
notification of processed MLME management frame

Parameters

struct net_device * dev network device

const u8 * buf authentication frame (header + body)

size_t len length of the frame data

Description

This function is called whenever an authentication, disassociation or deauthentication frame has been received and processed in station mode. After being asked to authenticate via cfg80211_ops::auth() the driver must call either this function or *cfg80211_auth_timeout()*. After being asked to associate via cfg80211_ops::assoc() the driver must call either this function or *cfg80211_auth_timeout()*. While connected, the driver must calls this for received and processed disassociation and deauthentication frames. If the frame couldn't be used because it was unprotected, the driver must call the function cfg80211_rx_unprot_mlme_mgmt() instead.

This function may sleep. The caller must hold the corresponding wdev's mutex.

void **cfg80211_auth_timeout**(struct net_device * *dev*, const u8 * *addr*)
notification of timed out authentication

Parameters

struct net_device * dev network device

const u8 * addr The MAC address of the device with which the authentication timed out

Description

This function may sleep. The caller must hold the corresponding wdev's mutex.

void **cfg80211_rx_assoc_resp**(struct net_device * *dev*, struct *cfg80211_bss* * *bss*, const u8 * *buf*, size_t *len*, int *uapsd_queues*)
notification of processed association response

Parameters

struct net_device * dev network device

struct cfg80211_bss * bss the BSS that association was requested with, ownership of the pointer moves to cfg80211 in this call

const u8 * buf authentication frame (header + body)

size_t len length of the frame data

int uapsd_queues bitmap of queues configured for uapsd. Same format as the AC bitmap in the QoS info field

Description

After being asked to associate via cfg80211_ops::assoc() the driver must call either this function or *cfg80211_auth_timeout()*.

This function may sleep. The caller must hold the corresponding wdev's mutex.

void **cfg80211_assoc_timeout**(struct net_device * *dev*, struct *cfg80211_bss* * *bss*)
notification of timed out association

Parameters

struct net_device * dev network device

struct cfg80211_bss * bss The BSS entry with which association timed out.

Description

This function may sleep. The caller must hold the corresponding wdev's mutex.

void **cfg80211_tx_mlme_mgmt**(struct net_device * *dev*, const u8 * *buf*, size_t *len*)
notification of transmitted deauth/disassoc frame

Parameters

struct net_device * dev network device

const u8 * buf 802.11 frame (header + body)

size_t len length of the frame data

Description

This function is called whenever deauthentication has been processed in station mode. This includes both received deauthentication frames and locally generated ones. This function may sleep. The caller must hold the corresponding wdev's mutex.

void **cfg80211_ibss_joined**(struct net_device * *dev*, const u8 * *bssid*, struct *ieee80211_channel* * *channel*, gfp_t *gfp*)
notify cfg80211 that device joined an IBSS

Parameters

struct net_device * dev network device

const u8 * bssid the BSSID of the IBSS joined

struct ieee80211_channel * channel the channel of the IBSS joined

gfp_t gfp allocation flags

Description

This function notifies cfg80211 that the device joined an IBSS or switched to a different BSSID. Before this function can be called, either a beacon has to have been received from the IBSS, or one of the cfg80211_inform_bss{,_frame} functions must have been called with the locally generated beacon – this guarantees that there is always a scan result for this IBSS. cfg80211 will handle the rest.

struct **cfg80211_connect_resp_params**
Connection response params

Definition

```
struct cfg80211_connect_resp_params {
  int status;
  const u8 *bssid;
  struct cfg80211_bss *bss;
  const u8 *req_ie;
  size_t req_ie_len;
  const u8 *resp_ie;
  size_t resp_ie_len;
  const u8 *fils_kek;
  size_t fils_kek_len;
  bool update_erp_next_seq_num;
  u16 fils_erp_next_seq_num;
  const u8 *pmk;
  size_t pmk_len;
  const u8 *pmkid;
  enum nl80211_timeout_reason timeout_reason;
};
```

Members

status Status code, WLAN_STATUS_SUCCESS for successful connection, use WLAN_STATUS_UNSPECIFIED_FAILURE if your device cannot give you the real status code for failures. If this call is used to report a failure due to a timeout (e.g., not receiving an Authentication frame from the AP) instead of an explicit rejection by the AP, -1 is used to indicate that this is a failure, but without a status code. **timeout_reason** is used to report the reason for the timeout in that case.

bssid The BSSID of the AP (may be NULL)

bss Entry of bss to which STA got connected to, can be obtained through cfg80211_get_bss() (may be NULL). Only one parameter among **bssid** and **bss** needs to be specified.

req_ie Association request IEs (may be NULL)

req_ie_len Association request IEs length

resp_ie Association response IEs (may be NULL)

resp_ie_len Association response IEs length

fils_kek KEK derived from a successful FILS connection (may be NULL)

fils_kek_len Length of **fils_kek** in octets

update_erp_next_seq_num Boolean value to specify whether the value in **fils_erp_next_seq_num** is valid.

fils_erp_next_seq_num The next sequence number to use in ERP message in FILS Authentication. This value should be specified irrespective of the status for a FILS connection.

pmk A new PMK if derived from a successful FILS connection (may be NULL).

pmk_len Length of **pmk** in octets

pmkid A new PMKID if derived from a successful FILS connection or the PMKID used for this FILS connection (may be NULL).

timeout_reason Reason for connection timeout. This is used when the connection fails due to a timeout instead of an explicit rejection from the AP. NL80211_TIMEOUT_UNSPECIFIED is used when the timeout reason is not known. This value is used only if **status** < 0 to indicate that the failure is due to a timeout and not due to explicit rejection by the AP. This value is ignored in other cases (**status** >= 0).

void **cfg80211_connect_done**(struct net_device ** dev*, struct *cfg80211_connect_resp_params* ** params*, gfp_t *gfp*)
notify cfg80211 of connection result

Parameters

struct net_device * dev network device

struct cfg80211_connect_resp_params * params connection response parameters

gfp_t gfp allocation flags

Description

It should be called by the underlying driver once execution of the connection request from connect() has been completed. This is similar to *cfg80211_connect_bss()*, but takes a structure pointer for connection response parameters. Only one of the functions among *cfg80211_connect_bss()*, *cfg80211_connect_result()*, *cfg80211_connect_timeout()*, and *cfg80211_connect_done()* should be called.

void **cfg80211_connect_result**(struct net_device **dev*, const u8 **bssid*, const u8 **req_ie*, size_t *req_ie_len*, const u8 * *resp_ie*, size_t *resp_ie_len*, u16 *status*, gfp_t *gfp*)
notify cfg80211 of connection result

Parameters

struct net_device * dev network device

const u8 * bssid the BSSID of the AP

const u8 * req_ie association request IEs (maybe be NULL)

size_t req_ie_len association request IEs length

const u8 * resp_ie association response IEs (may be NULL)

size_t resp_ie_len assoc response IEs length

u16 status status code, WLAN_STATUS_SUCCESS for successful connection, use WLAN_STATUS_UNSPECIFIED_FAILURE if your device cannot give you the real status code for failures.

gfp_t gfp allocation flags

Description

It should be called by the underlying driver once execution of the connection request from connect() has been completed. This is similar to *cfg80211_connect_bss()* which allows the exact bss entry to be specified. Only one of the functions among *cfg80211_connect_bss()*, *cfg80211_connect_result()*, *cfg80211_connect_timeout()*, and *cfg80211_connect_done()* should be called.

void **cfg80211_connect_bss**(struct net_device **dev*, const u8 **bssid*, struct *cfg80211_bss* * *bss*, const u8 **req_ie*, size_t *req_ie_len*, const u8 **resp_ie*, size_t *resp_ie_len*, int *status*, gfp_t *gfp*, enum nl80211_timeout_reason *timeout_reason*)
notify cfg80211 of connection result

Parameters

struct net_device * dev network device

const u8 * bssid the BSSID of the AP

struct cfg80211_bss * bss entry of bss to which STA got connected to, can be obtained through cfg80211_get_bss (may be NULL)

const u8 * req_ie association request IEs (maybe be NULL)

size_t req_ie_len association request IEs length

const u8 * resp_ie association response IEs (may be NULL)

size_t resp_ie_len assoc response IEs length

int status status code, `WLAN_STATUS_SUCCESS` for successful connection, use `WLAN_STATUS_UNSPECIFIED_FAILURE` if your device cannot give you the real status code for failures. If this call is used to report a failure due to a timeout (e.g., not receiving an Authentication frame from the AP) instead of an explicit rejection by the AP, -1 is used to indicate that this is a failure, but without a status code. **timeout_reason** is used to report the reason for the timeout in that case.

gfp_t gfp allocation flags

enum nl80211_timeout_reason timeout_reason reason for connection timeout. This is used when the connection fails due to a timeout instead of an explicit rejection from the AP. `NL80211_TIMEOUT_UNSPECIFIED` is used when the timeout reason is not known. This value is used only if **status** < 0 to indicate that the failure is due to a timeout and not due to explicit rejection by the AP. This value is ignored in other cases (**status** >= 0).

Description

It should be called by the underlying driver once execution of the connection request from `connect()` has been completed. This is similar to *cfg80211_connect_result()*, but with the option of identifying the exact bss entry for the connection. Only one of the functions among *cfg80211_connect_bss()*, *cfg80211_connect_result()*, *cfg80211_connect_timeout()*, and *cfg80211_connect_done()* should be called.

void **cfg80211_connect_timeout**(struct net_device * *dev*, const u8 * *bssid*, const u8 * *req_ie*, size_t *req_ie_len*, gfp_t *gfp*, enum nl80211_timeout_reason *timeout_reason*)
notify cfg80211 of connection timeout

Parameters

struct net_device * dev network device

const u8 * bssid the BSSID of the AP

const u8 * req_ie association request IEs (maybe be `NULL`)

size_t req_ie_len association request IEs length

gfp_t gfp allocation flags

enum nl80211_timeout_reason timeout_reason reason for connection timeout.

Description

It should be called by the underlying driver whenever `connect()` has failed in a sequence where no explicit authentication/association rejection was received from the AP. This could happen, e.g., due to not being able to send out the Authentication or Association Request frame or timing out while waiting for the response. Only one of the functions among *cfg80211_connect_bss()*, *cfg80211_connect_result()*, *cfg80211_connect_timeout()*, and *cfg80211_connect_done()* should be called.

void **cfg80211_roamed**(struct net_device * *dev*, struct cfg80211_roam_info * *info*, gfp_t *gfp*)
notify cfg80211 of roaming

Parameters

struct net_device * dev network device

struct cfg80211_roam_info * info information about the new BSS. struct `cfg80211_roam_info`.

gfp_t gfp allocation flags

Description

This function may be called with the driver passing either the BSSID of the new AP or passing the bss entry to avoid a race in timeout of the bss entry. It should be called by the underlying driver whenever it roamed from one AP to another while connected. Drivers which have roaming implemented in firmware should pass the bss entry to avoid a race in bss entry timeout where the bss entry of the new AP is seen in the driver, but gets timed out by the time it is accessed in `__cfg80211_roamed()` due to delay in scheduling rdev->event_work. In case of any failures, the reference is released either in `cfg80211_roamed()` or in `__cfg80211_romed()`, Otherwise, it will be released while diconneting from the current bss.

void **cfg80211_disconnected**(struct net_device * *dev*, u16 *reason*, const u8 * *ie*, size_t *ie_len*, bool *locally_generated*, gfp_t *gfp*)
notify cfg80211 that connection was dropped

Parameters

struct net_device * dev network device

u16 reason reason code for the disconnection, set it to 0 if unknown

const u8 * ie information elements of the deauth/disassoc frame (may be NULL)

size_t ie_len length of IEs

bool locally_generated disconnection was requested locally

gfp_t gfp allocation flags

Description

After it calls this function, the driver should enter an idle state and not try to connect to any AP any more.

void **cfg80211_ready_on_channel**(struct *wireless_dev* * *wdev*, u64 *cookie*, struct *ieee80211_channel* * *chan*, unsigned int *duration*, gfp_t *gfp*)
notification of remain_on_channel start

Parameters

struct wireless_dev * wdev wireless device

u64 cookie the request cookie

struct ieee80211_channel * chan The current channel (from remain_on_channel request)

unsigned int duration Duration in milliseconds that the driver intents to remain on the channel

gfp_t gfp allocation flags

void **cfg80211_remain_on_channel_expired**(struct *wireless_dev* * *wdev*, u64 *cookie*, struct *ieee80211_channel* * *chan*, gfp_t *gfp*)
remain_on_channel duration expired

Parameters

struct wireless_dev * wdev wireless device

u64 cookie the request cookie

struct ieee80211_channel * chan The current channel (from remain_on_channel request)

gfp_t gfp allocation flags

void **cfg80211_new_sta**(struct net_device * *dev*, const u8 * *mac_addr*, struct *station_info* * *sinfo*, gfp_t *gfp*)
notify userspace about station

Parameters

struct net_device * dev the netdev

const u8 * mac_addr the station's address

struct station_info * sinfo the station information

gfp_t gfp allocation flags

bool **cfg80211_rx_mgmt**(struct *wireless_dev* * *wdev*, int *freq*, int *sig_dbm*, const u8 * *buf*, size_t *len*, u32 *flags*)
notification of received, unprocessed management frame

Parameters

struct wireless_dev * wdev wireless device receiving the frame

int freq Frequency on which the frame was received in MHz

int sig_dbm signal strength in dBm, or 0 if unknown

const u8 * buf Management frame (header + body)

size_t len length of the frame data

u32 flags flags, as defined in enum nl80211_rxmgmt_flags

Description

This function is called whenever an Action frame is received for a station mode interface, but is not processed in kernel.

Return

`true` if a user space application has registered for this frame. For action frames, that makes it responsible for rejecting unrecognized action frames; `false` otherwise, in which case for action frames the driver is responsible for rejecting the frame.

void **cfg80211_mgmt_tx_status**(struct *wireless_dev* * *wdev*, u64 *cookie*, const u8 * *buf*, size_t *len*, bool *ack*, gfp_t *gfp*)
notification of TX status for management frame

Parameters

struct wireless_dev * wdev wireless device receiving the frame

u64 cookie Cookie returned by cfg80211_ops::`mgmt_tx()`

const u8 * buf Management frame (header + body)

size_t len length of the frame data

bool ack Whether frame was acknowledged

gfp_t gfp context flags

Description

This function is called whenever a management frame was requested to be transmitted with cfg80211_ops::`mgmt_tx()` to report the TX status of the transmission attempt.

void **cfg80211_cqm_rssi_notify**(struct net_device * *dev*, enum nl80211_cqm_rssi_threshold_event *rssi_event*, s32 *rssi_level*, gfp_t *gfp*)
connection quality monitoring rssi event

Parameters

struct net_device * dev network device

enum nl80211_cqm_rssi_threshold_event rssi_event the triggered RSSI event

s32 rssi_level new RSSI level value or 0 if not available

gfp_t gfp context flags

Description

This function is called when a configured connection quality monitoring rssi threshold reached event occurs.

void **cfg80211_cqm_pktloss_notify**(struct net_device * *dev*, const u8 * *peer*, u32 *num_packets*, gfp_t *gfp*)
notify userspace about packetloss to peer

Parameters

struct net_device * dev network device

const u8 * peer peer's MAC address

u32 num_packets how many packets were lost – should be a fixed threshold but probably no less than maybe 50, or maybe a throughput dependent threshold (to account for temporary interference)

gfp_t gfp context flags

void **cfg80211_michael_mic_failure**(struct net_device * *dev*, const u8 * *addr*, enum nl80211_key_type *key_type*, int *key_id*, const u8 * *tsc*, gfp_t *gfp*)
notification of Michael MIC failure (TKIP)

Parameters

struct net_device * dev network device

const u8 * addr The source MAC address of the frame

enum nl80211_key_type key_type The key type that the received frame used

int key_id Key identifier (0..3). Can be -1 if missing.

const u8 * tsc The TSC value of the frame that generated the MIC failure (6 octets)

gfp_t gfp allocation flags

Description

This function is called whenever the local MAC detects a MIC failure in a received frame. This matches with MLME-MICHAELMICFAILURE.:c:func:*indication()* primitive.

Scanning and BSS list handling

The scanning process itself is fairly simple, but cfg80211 offers quite a bit of helper functionality. To start a scan, the scan operation will be invoked with a scan definition. This scan definition contains the channels to scan, and the SSIDs to send probe requests for (including the wildcard, if desired). A passive scan is indicated by having no SSIDs to probe. Additionally, a scan request may contain extra information elements that should be added to the probe request. The IEs are guaranteed to be well-formed, and will not exceed the maximum length the driver advertised in the wiphy structure.

When scanning finds a BSS, cfg80211 needs to be notified of that, because it is responsible for maintaining the BSS list; the driver should not maintain a list itself. For this notification, various functions exist.

Since drivers do not maintain a BSS list, there are also a number of functions to search for a BSS and obtain information about it from the BSS structure cfg80211 maintains. The BSS list is also made available to userspace.

struct **cfg80211_ssid**
SSID description

Definition

```
struct cfg80211_ssid {
  u8 ssid[IEEE80211_MAX_SSID_LEN];
  u8 ssid_len;
};
```

Members

ssid the SSID

ssid_len length of the ssid

struct **cfg80211_scan_request**
scan request description

Definition

```
struct cfg80211_scan_request {
  struct cfg80211_ssid *ssids;
  int n_ssids;
  u32 n_channels;
  enum nl80211_bss_scan_width scan_width;
  const u8 *ie;
  size_t ie_len;
  u16 duration;
  bool duration_mandatory;
  u32 flags;
  u32 rates[NUM_NL80211_BANDS];
  struct wireless_dev *wdev;
  u8 mac_addr[ETH_ALEN] ;
  u8 mac_addr_mask[ETH_ALEN] ;
  u8 bssid[ETH_ALEN] ;
  struct wiphy *wiphy;
  unsigned long scan_start;
  struct cfg80211_scan_info info;
  bool notified;
  bool no_cck;
  struct ieee80211_channel *channels[0];
};
```

Members

`ssids` SSIDs to scan for (active scan only)

`n_ssids` number of SSIDs

`n_channels` total number of channels to scan

`scan_width` channel width for scanning

`ie` optional information element(s) to add into Probe Request or `NULL`

`ie_len` length of ie in octets

`duration` how long to listen on each channel, in TUs. If `duration_mandatory` is not set, this is the maximum dwell time and the actual dwell time may be shorter.

`duration_mandatory` if set, the scan duration must be as specified by the `duration` field.

`flags` bit field of flags controlling operation

`rates` bitmap of rates to advertise for each band

`wdev` the wireless device to scan for

`mac_addr` MAC address used with randomisation

`mac_addr_mask` MAC address mask used with randomisation, bits that are 0 in the mask should be randomised, bits that are 1 should be taken from the **mac_addr**

`bssid` BSSID to scan for (most commonly, the wildcard BSSID)

`wiphy` the wiphy this was for

`scan_start` time (in jiffies) when the scan started

`info` (internal) information about completed scan

`notified` (internal) scan request was notified as done or aborted

`no_cck` used to send probe requests at non CCK rate in 2GHz band

`channels` channels to scan on.

void **`cfg80211_scan_done`**(struct *cfg80211_scan_request* * *request*, struct cfg80211_scan_info * *info*)
notify that scan finished

Parameters

struct cfg80211_scan_request * request the corresponding scan request

struct cfg80211_scan_info * info information about the completed scan

struct **cfg80211_bss**
BSS description

Definition

```
struct cfg80211_bss {
  struct ieee80211_channel *channel;
  enum nl80211_bss_scan_width scan_width;
  const struct cfg80211_bss_ies __rcu *ies;
  const struct cfg80211_bss_ies __rcu *beacon_ies;
  const struct cfg80211_bss_ies __rcu *proberesp_ies;
  struct cfg80211_bss *hidden_beacon_bss;
  s32 signal;
  u16 beacon_interval;
  u16 capability;
  u8 bssid[ETH_ALEN];
  u8 chains;
  s8 chain_signal[IEEE80211_MAX_CHAINS];
  u8 priv[0] ;
};
```

Members

channel channel this BSS is on

scan_width width of the control channel

ies the information elements (Note that there is no guarantee that these are well-formed!); this is a pointer to either the beacon_ies or proberesp_ies depending on whether Probe Response frame has been received. It is always non-NULL.

beacon_ies the information elements from the last Beacon frame (implementation note: if **hidden_beacon_bss** is set this struct doesn't own the beacon_ies, but they're just pointers to the ones from the **hidden_beacon_bss** struct)

proberesp_ies the information elements from the last Probe Response frame

hidden_beacon_bss in case this BSS struct represents a probe response from a BSS that hides the SSID in its beacon, this points to the BSS struct that holds the beacon data. **beacon_ies** is still valid, of course, and points to the same data as hidden_beacon_bss->beacon_ies in that case.

signal signal strength value (type depends on the wiphy's signal_type)

beacon_interval the beacon interval as from the frame

capability the capability field in host byte order

bssid BSSID of the BSS

chains bitmask for filled values in **chain_signal**.

chain_signal per-chain signal strength of last received BSS in dBm.

priv private area for driver use, has at least wiphy->bss_priv_size bytes

Description

This structure describes a BSS (which may also be a mesh network) for use in scan results and similar.

struct **cfg80211_inform_bss**
BSS inform data

Definition

```
struct cfg80211_inform_bss {
  struct ieee80211_channel *chan;
  enum nl80211_bss_scan_width scan_width;
  s32 signal;
  u64 boottime_ns;
  u64 parent_tsf;
  u8 parent_bssid[ETH_ALEN] ;
  u8 chains;
  s8 chain_signal[IEEE80211_MAX_CHAINS];
};
```

Members

chan channel the frame was received on

scan_width scan width that was used

signal signal strength value, according to the wiphy's signal type

boottime_ns timestamp (CLOCK_BOOTTIME) when the information was received; should match the time when the frame was actually received by the device (not just by the host, in case it was buffered on the device) and be accurate to about 10ms. If the frame isn't buffered, just passing the return value of `ktime_get_boot_ns()` is likely appropriate.

parent_tsf the time at the start of reception of the first octet of the timestamp field of the frame. The time is the TSF of the BSS specified by `parent_bssid`.

parent_bssid the BSS according to which `parent_tsf` is set. This is set to the BSS that requested the scan in which the beacon/probe was received.

chains bitmask for filled values in **chain_signal**.

chain_signal per-chain signal strength of last received BSS in dBm.

struct *cfg80211_bss* * **cfg80211_inform_bss_frame_data**(struct *wiphy* * *wiphy*, struct *cfg80211_inform_bss* * *data*, struct ieee80211_mgmt * *mgmt*, size_t *len*, gfp_t *gfp*)

inform cfg80211 of a received BSS frame

Parameters

struct wiphy * wiphy the wiphy reporting the BSS

struct cfg80211_inform_bss * data the BSS metadata

struct ieee80211_mgmt * mgmt the management frame (probe response or beacon)

size_t len length of the management frame

gfp_t gfp context flags

Description

This informs cfg80211 that BSS information was found and the BSS should be updated/added.

Return

A referenced struct, must be released with `cfg80211_put_bss()`! Or NULL on error.

struct *cfg80211_bss* * **cfg80211_inform_bss_data**(struct *wiphy* * *wiphy*, struct *cfg80211_inform_bss* * *data*, enum cfg80211_bss_frame_type *ftype*, const u8 * *bssid*, u64 *tsf*, u16 *capability*, u16 *beacon_interval*, const u8 * *ie*, size_t *ielen*, gfp_t *gfp*)

inform cfg80211 of a new BSS

Parameters

struct wiphy * wiphy the wiphy reporting the BSS

struct cfg80211_inform_bss * data the BSS metadata

enum cfg80211_bss_frame_type ftype frame type (if known)

const u8 * bssid the BSSID of the BSS

u64 tsf the TSF sent by the peer in the beacon/probe response (or 0)

u16 capability the capability field sent by the peer

u16 beacon_interval the beacon interval announced by the peer

const u8 * ie additional IEs sent by the peer

size_t ielen length of the additional IEs

gfp_t gfp context flags

Description

This informs cfg80211 that BSS information was found and the BSS should be updated/added.

Return

A referenced struct, must be released with `cfg80211_put_bss()`! Or NULL on error.

void **cfg80211_unlink_bss**(struct *wiphy* * *wiphy*, struct *cfg80211_bss* * *bss*)
    unlink BSS from internal data structures

Parameters

struct wiphy * wiphy the wiphy

struct cfg80211_bss * bss the bss to remove

Description

This function removes the given BSS from the internal data structures thereby making it no longer show up in scan results etc. Use this function when you detect a BSS is gone. Normally BSSes will also time out, so it is not necessary to use this function at all.

const u8 * **cfg80211_find_ie**(u8 *eid*, const u8 * *ies*, int *len*)
    find information element in data

Parameters

u8 eid element ID

const u8 * ies data consisting of IEs

int len length of data

Return

NULL if the element ID could not be found or if the element is invalid (claims to be longer than the given data), or a pointer to the first byte of the requested element, that is the byte containing the element ID.

Note

There are no checks on the element length other than having to fit into the given data.

const u8 * **ieee80211_bss_get_ie**(struct *cfg80211_bss* * *bss*, u8 *ie*)
    find IE with given ID

Parameters

struct cfg80211_bss * bss the bss to search

u8 ie the IE ID

Description

Note that the return value is an RCU-protected pointer, so `rcu_read_lock()` must be held when calling this function.

Return

`NULL` if not found.

Utility functions

cfg80211 offers a number of utility functions that can be useful.

int **ieee80211_channel_to_frequency**(int *chan*, enum nl80211_band *band*)
convert channel number to frequency

Parameters

int chan channel number

enum nl80211_band band band, necessary due to channel number overlap

Return

The corresponding frequency (in MHz), or 0 if the conversion failed.

int **ieee80211_frequency_to_channel**(int *freq*)
convert frequency to channel number

Parameters

int freq center frequency

Return

The corresponding channel, or 0 if the conversion failed.

struct *ieee80211_channel* * **ieee80211_get_channel**(struct *wiphy* * *wiphy*, int *freq*)
get channel struct from wiphy for specified frequency

Parameters

struct wiphy * wiphy the struct wiphy to get the channel for

int freq the center frequency of the channel

Return

The channel struct from **wiphy** at **freq**.

struct *ieee80211_rate* * **ieee80211_get_response_rate**(struct *ieee80211_supported_band* * *sband*, u32 *basic_rates*, int *bitrate*)
get basic rate for a given rate

Parameters

struct ieee80211_supported_band * sband the band to look for rates in

u32 basic_rates bitmap of basic rates

int bitrate the bitrate for which to find the basic rate

Return

The basic rate corresponding to a given bitrate, that is the next lower bitrate contained in the basic rate map, which is, for this function, given as a bitmap of indices of rates in the band's bitrate table.

unsigned int __attribute_const__ **ieee80211_hdrlen**(__le16 *fc*)
get header length in bytes from frame control

Parameters

__le16 fc frame control field in little-endian format

Return

The header length in bytes.

unsigned int **ieee80211_get_hdrlen_from_skb**(const struct sk_buff * *skb*)
get header length from data

Parameters

const struct sk_buff * skb the frame

Description

Given an skb with a raw 802.11 header at the data pointer this function returns the 802.11 header length.

Return

The 802.11 header length in bytes (not including encryption headers). Or 0 if the data in the sk_buff is too short to contain a valid 802.11 header.

struct **ieee80211_radiotap_iterator**
tracks walk thru present radiotap args

Definition

```
struct ieee80211_radiotap_iterator {
  struct ieee80211_radiotap_header *_rtheader;
  const struct ieee80211_radiotap_vendor_namespaces *_vns;
  const struct ieee80211_radiotap_namespace *current_namespace;
  unsigned char *_arg, *_next_ns_data;
  __le32 *_next_bitmap;
  unsigned char *this_arg;
  int this_arg_index;
  int this_arg_size;
  int is_radiotap_ns;
  int _max_length;
  int _arg_index;
  uint32_t _bitmap_shifter;
  int _reset_on_ext;
};
```

Members

_rtheader pointer to the radiotap header we are walking through

_vns vendor namespace definitions

current_namespace pointer to the current namespace definition (or internally NULL if the current namespace is unknown)

_arg next argument pointer

_next_ns_data beginning of the next namespace's data

_next_bitmap internal pointer to next present u32

this_arg pointer to current radiotap arg; it is valid after each call to ieee80211_radiotap_iterator_next() but also after ieee80211_radiotap_iterator_init() where it will point to the beginning of the actual data portion

this_arg_index index of current arg, valid after each successful call to ieee80211_radiotap_iterator_next()

this_arg_size length of the current arg, for convenience

is_radiotap_ns indicates whether the current namespace is the default radiotap namespace or not

_max_length length of radiotap header in cpu byte ordering

`_arg_index` next argument index

`_bitmap_shifter` internal shifter for curr u32 bitmap, b0 set == arg present

`_reset_on_ext` internal; reset the arg index to 0 when going to the next bitmap word

Description

Describes the radiotap parser state. Fields prefixed with an underscore must not be used by users of the parser, only by the parser internally.

Data path helpers

In addition to generic utilities, cfg80211 also offers functions that help implement the data path for devices that do not do the 802.11/802.3 conversion on the device.

int **`ieee80211_data_to_8023`**(struct sk_buff * *skb*, const u8 * *addr*, enum nl80211_iftype *iftype*)
convert an 802.11 data frame to 802.3

Parameters

`struct sk_buff * skb` the 802.11 data frame

`const u8 * addr` the device MAC address

`enum nl80211_iftype iftype` the virtual interface type

Return

0 on success. Non-zero on error.

void **`ieee80211_amsdu_to_8023s`**(struct sk_buff * *skb*, struct sk_buff_head * *list*, const u8 * *addr*, enum nl80211_iftype *iftype*, const unsigned int *extra_headroom*, const u8 * *check_da*, const u8 * *check_sa*)
decode an IEEE 802.11n A-MSDU frame

Parameters

`struct sk_buff * skb` The input A-MSDU frame without any headers.

`struct sk_buff_head * list` The output list of 802.3 frames. It must be allocated and initialized by by the caller.

`const u8 * addr` The device MAC address.

`enum nl80211_iftype iftype` The device interface type.

`const unsigned int extra_headroom` The hardware extra headroom for SKBs in the **list**.

`const u8 * check_da` DA to check in the inner ethernet header, or NULL

`const u8 * check_sa` SA to check in the inner ethernet header, or NULL

Description

Decode an IEEE 802.11 A-MSDU and convert it to a list of 802.3 frames. The **list** will be empty if the decode fails. The **skb** must be fully header-less before being passed in here; it is freed in this function.

unsigned int **`cfg80211_classify8021d`**(struct sk_buff * *skb*, struct cfg80211_qos_map * *qos_map*)
determine the 802.1p/1d tag for a data frame

Parameters

`struct sk_buff * skb` the data frame

`struct cfg80211_qos_map * qos_map` Interworking QoS mapping or `NULL` if not in use

Return

The 802.1p/1d tag.

Regulatory enforcement infrastructure

TODO

int **regulatory_hint**(struct *wiphy* * *wiphy*, const char * *alpha2*)
driver hint to the wireless core a regulatory domain

Parameters

struct wiphy * wiphy the wireless device giving the hint (used only for reporting conflicts)

const char * alpha2 the ISO/IEC 3166 alpha2 the driver claims its regulatory domain should be in. If **rd** is set this should be NULL. Note that if you set this to NULL you should still set rd->alpha2 to some accepted alpha2.

Description

Wireless drivers can use this function to hint to the wireless core what it believes should be the current regulatory domain by giving it an ISO/IEC 3166 alpha2 country code it knows its regulatory domain should be in or by providing a completely build regulatory domain. If the driver provides an ISO/IEC 3166 alpha2 userspace will be queried for a regulatory domain structure for the respective country.

The wiphy must have been registered to cfg80211 prior to this call. For cfg80211 drivers this means you must first use *wiphy_register()*, for mac80211 drivers you must first use *ieee80211_register_hw()*.

Drivers should check the return value, its possible you can get an -ENOMEM.

Return

0 on success. -ENOMEM.

void **wiphy_apply_custom_regulatory**(struct *wiphy* * *wiphy*, const struct ieee80211_regdomain * *regd*)
apply a custom driver regulatory domain

Parameters

struct wiphy * wiphy the wireless device we want to process the regulatory domain on

const struct ieee80211_regdomain * regd the custom regulatory domain to use for this wiphy

Description

Drivers can sometimes have custom regulatory domains which do not apply to a specific country. Drivers can use this to apply such custom regulatory domains. This routine must be called prior to wiphy registration. The custom regulatory domain will be trusted completely and as such previous default channel settings will be disregarded. If no rule is found for a channel on the regulatory domain the channel will be disabled. Drivers using this for a wiphy should also set the wiphy flag REGULATORY_CUSTOM_REG or cfg80211 will set it for the wiphy that called this helper.

const struct ieee80211_reg_rule * **freq_reg_info**(struct *wiphy* * *wiphy*, u32 *center_freq*)
get regulatory information for the given frequency

Parameters

struct wiphy * wiphy the wiphy for which we want to process this rule for

u32 center_freq Frequency in KHz for which we want regulatory information for

Description

Use this function to get the regulatory rule for a specific frequency on a given wireless device. If the device has a specific regulatory domain it wants to follow we respect that unless a country IE has been received and processed already.

Return

A valid pointer, or, when an error occurs, for example if no rule can be found, the return value is encoded using ERR_PTR(). Use IS_ERR() to check and PTR_ERR() to obtain the numeric return value. The numeric return value will be -ERANGE if we determine the given center_freq does not even have a regulatory rule for a frequency range in the center_freq's band. See `freq_in_rule_band()` for our current definition of a band – this is purely subjective and right now it's 802.11 specific.

RFkill integration

RFkill integration in cfg80211 is almost invisible to drivers, as cfg80211 automatically registers an rfkill instance for each wireless device it knows about. Soft kill is also translated into disconnecting and turning all interfaces off, drivers are expected to turn off the device when all interfaces are down.

However, devices may have a hard RFkill line, in which case they also need to interact with the rfkill subsystem, via cfg80211. They can do this with a few helper functions documented here.

void **wiphy_rfkill_set_hw_state**(struct *wiphy* * *wiphy*, bool *blocked*)
    notify cfg80211 about hw block state

Parameters

struct wiphy * wiphy the wiphy

bool blocked block status

void **wiphy_rfkill_start_polling**(struct *wiphy* * *wiphy*)
    start polling rfkill

Parameters

struct wiphy * wiphy the wiphy

void **wiphy_rfkill_stop_polling**(struct *wiphy* * *wiphy*)
    stop polling rfkill

Parameters

struct wiphy * wiphy the wiphy

Test mode

Test mode is a set of utility functions to allow drivers to interact with driver-specific tools to aid, for instance, factory programming.

This chapter describes how drivers interact with it, for more information see the nl80211 book's chapter on it.

struct sk_buff * **cfg80211_testmode_alloc_reply_skb**(struct *wiphy* * *wiphy*, int *approxlen*)
    allocate testmode reply

Parameters

struct wiphy * wiphy the wiphy

int approxlen an upper bound of the length of the data that will be put into the skb

Description

This function allocates and pre-fills an skb for a reply to the testmode command. Since it is intended for a reply, calling it outside of the **testmode_cmd** operation is invalid.

The returned skb is pre-filled with the wiphy index and set up in a way that any data that is put into the skb (with `skb_put()`, `nla_put()` or similar) will end up being within the `NL80211_ATTR_TESTDATA` attribute, so all that needs to be done with the skb is adding data for the corresponding userspace tool which can then read that data out of the testdata attribute. You must not modify the skb in any other way.

When done, call *`cfg80211_testmode_reply()`* with the skb and return its error code as the result of the **testmode_cmd** operation.

Return

An allocated and pre-filled skb. NULL if any errors happen.

int **cfg80211_testmode_reply**(struct sk_buff * *skb*)
send the reply skb

Parameters

struct sk_buff * skb The skb, must have been allocated with *cfg80211_testmode_alloc_reply_skb()*

Description

Since calling this function will usually be the last thing before returning from the **testmode_cmd** you should return the error code. Note that this function consumes the skb regardless of the return value.

Return

An error code or 0 on success.

struct sk_buff * **cfg80211_testmode_alloc_event_skb**(struct *wiphy* * *wiphy*, int *approxlen*, gfp_t *gfp*)
allocate testmode event

Parameters

struct wiphy * wiphy the wiphy

int approxlen an upper bound of the length of the data that will be put into the skb

gfp_t gfp allocation flags

Description

This function allocates and pre-fills an skb for an event on the testmode multicast group.

The returned skb is set up in the same way as with *cfg80211_testmode_alloc_reply_skb()* but prepared for an event. As there, you should simply add data to it that will then end up in the NL80211_ATTR_TESTDATA attribute. Again, you must not modify the skb in any other way.

When done filling the skb, call *cfg80211_testmode_event()* with the skb to send the event.

Return

An allocated and pre-filled skb. NULL if any errors happen.

void **cfg80211_testmode_event**(struct sk_buff * *skb*, gfp_t *gfp*)
send the event

Parameters

struct sk_buff * skb The skb, must have been allocated with *cfg80211_testmode_alloc_event_skb()*

gfp_t gfp allocation flags

Description

This function sends the given **skb**, which must have been allocated by *cfg80211_testmode_alloc_event_skb()*, as an event. It always consumes it.

mac80211 subsystem (basics)

You should read and understand the information contained within this part of the book while implementing a mac80211 driver. In some chapters, advanced usage is noted, those may be skipped if this isn't needed.

This part of the book only covers station and monitor mode functionality, additional information required to implement the other modes is covered in the second part of the book.

Basic hardware handling

TBD

This chapter shall contain information on getting a hw struct allocated and registered with mac80211.

Since it is required to allocate rates/modes before registering a hw struct, this chapter shall also contain information on setting up the rate/mode structs.

Additionally, some discussion about the callbacks and the general programming model should be in here, including the definition of ieee80211_ops which will be referred to a lot.

Finally, a discussion of hardware capabilities should be done with references to other parts of the book.

struct **ieee80211_hw**
    hardware information and state

Definition

```
struct ieee80211_hw {
  struct ieee80211_conf conf;
  struct wiphy *wiphy;
  const char *rate_control_algorithm;
  void *priv;
  unsigned long flags[BITS_TO_LONGS(NUM_IEEE80211_HW_FLAGS)];
  unsigned int extra_tx_headroom;
  unsigned int extra_beacon_tailroom;
  int vif_data_size;
  int sta_data_size;
  int chanctx_data_size;
  int txq_data_size;
  u16 queues;
  u16 max_listen_interval;
  s8 max_signal;
  u8 max_rates;
  u8 max_report_rates;
  u8 max_rate_tries;
  u8 max_rx_aggregation_subframes;
  u8 max_tx_aggregation_subframes;
  u8 max_tx_fragments;
  u8 offchannel_tx_hw_queue;
  u8 radiotap_mcs_details;
  u16 radiotap_vht_details;
  struct {
    int units_pos;
    s16 accuracy;
  } radiotap_timestamp;
  netdev_features_t netdev_features;
  u8 uapsd_queues;
  u8 uapsd_max_sp_len;
  u8 n_cipher_schemes;
  const struct ieee80211_cipher_scheme *cipher_schemes;
  u8 max_nan_de_entries;
};
```

Members

conf *struct ieee80211_conf*, device configuration, don't use.

wiphy This points to the *struct wiphy* allocated for this 802.11 PHY. You must fill in the **perm_addr** and **dev** members of this structure using *SET_IEEE80211_DEV()* and *SET_IEEE80211_PERM_ADDR()*. Additionally, all supported bands (with channels, bitrates) are registered here.

rate_control_algorithm rate control algorithm for this hardware. If unset (NULL), the default algorithm will be used. Must be set before calling *ieee80211_register_hw()*.

priv pointer to private area that was allocated for driver use along with this structure.

flags hardware flags, see *enum ieee80211_hw_flags*.

extra_tx_headroom headroom to reserve in each transmit skb for use by the driver (e.g. for transmit headers.)

extra_beacon_tailroom tailroom to reserve in each beacon tx skb. Can be used by drivers to add extra IEs.

vif_data_size size (in bytes) of the drv_priv data area within *struct ieee80211_vif*.

sta_data_size size (in bytes) of the drv_priv data area within *struct ieee80211_sta*.

chanctx_data_size size (in bytes) of the drv_priv data area within `struct ieee80211_chanctx_conf`.

txq_data_size size (in bytes) of the drv_priv data area within **struct** ieee80211_txq.

queues number of available hardware transmit queues for data packets. WMM/QoS requires at least four, these queues need to have configurable access parameters.

max_listen_interval max listen interval in units of beacon interval that HW supports

max_signal Maximum value for signal (rssi) in RX information, used only when **IEEE80211_HW_SIGNAL_UNSPEC** or **IEEE80211_HW_SIGNAL_DB**

max_rates maximum number of alternate rate retry stages the hw can handle.

max_report_rates maximum number of alternate rate retry stages the hw can report back.

max_rate_tries maximum number of tries for each stage

max_rx_aggregation_subframes maximum buffer size (number of sub-frames) to be used for A-MPDU block ack receiver aggregation. This is only relevant if the device has restrictions on the number of subframes, if it relies on mac80211 to do reordering it shouldn't be set.

max_tx_aggregation_subframes maximum number of subframes in an aggregate an HT driver will transmit. Though ADDBA will advertise a constant value of 64 as some older APs can crash if the window size is smaller (an example is LinkSys WRT120N with FW v1.0.07 build 002 Jun 18 2012).

max_tx_fragments maximum number of tx buffers per (A)-MSDU, sum of 1 + skb_shinfo(skb)->nr_frags for each skb in the frag_list.

offchannel_tx_hw_queue HW queue ID to use for offchannel TX (if `IEEE80211_HW_QUEUE_CONTROL` is set)

radiotap_mcs_details lists which MCS information can the HW reports, by default it is set to _MCS, _GI and _BW but doesn't include _FMT. Use `IEEE80211_RADIOTAP_MCS_HAVE_*` values, only adding _BW is supported today.

radiotap_vht_details lists which VHT MCS information the HW reports, the default is _GI | _BANDWIDTH. Use the `IEEE80211_RADIOTAP_VHT_KNOWN_*` values.

radiotap_timestamp Information for the radiotap timestamp field; if the 'units_pos' member is set to a non-negative value it must be set to a combination of a IEEE80211_RADIOTAP_TIMESTAMP_UNIT_* and a IEEE80211_RADIOTAP_TIMESTAMP_SPOS_* value, and then the timestamp field will be added and populated from the *struct ieee80211_rx_status* device_timestamp. If the 'accuracy' member is non-negative, it's put into the accuracy radiotap field and the accuracy known flag is set.

netdev_features netdev features to be set in each netdev created from this HW. Note that not all features are usable with mac80211, other features will be rejected during HW registration.

uapsd_queues This bitmap is included in (re)association frame to indicate for each access category if it is uAPSD trigger-enabled and delivery- enabled. Use IEEE80211_WMM_IE_STA_QOSINFO_AC_* to set this bitmap. Each bit corresponds to different AC. Value '1' in specific bit means that corresponding AC is both trigger- and delivery-enabled. '0' means neither enabled.

uapsd_max_sp_len maximum number of total buffered frames the WMM AP may deliver to a WMM STA during any Service Period triggered by the WMM STA. Use IEEE80211_WMM_IE_STA_QOSINFO_SP_* for correct values.

n_cipher_schemes a size of an array of cipher schemes definitions.

cipher_schemes a pointer to an array of cipher scheme definitions supported by HW.

max_nan_de_entries maximum number of NAN DE functions supported by the device.

Description

This structure contains the configuration and hardware information for an 802.11 PHY.

enum **ieee80211_hw_flags**
hardware flags

Constants

IEEE80211_HW_HAS_RATE_CONTROL The hardware or firmware includes rate control, and cannot be controlled by the stack. As such, no rate control algorithm should be instantiated, and the TX rate reported to userspace will be taken from the TX status instead of the rate control algorithm. Note that this requires that the driver implement a number of callbacks so it has the correct information, it needs to have the **set_rts_threshold** callback and must look at the BSS config **use_cts_prot** for G/N protection, **use_short_slot** for slot timing in 2.4 GHz and **use_short_preamble** for preambles for CCK frames.

IEEE80211_HW_RX_INCLUDES_FCS Indicates that received frames passed to the stack include the FCS at the end.

IEEE80211_HW_HOST_BROADCAST_PS_BUFFERING Some wireless LAN chipsets buffer broadcast/multicast frames for power saving stations in the hardware/firmware and others rely on the host system for such buffering. This option is used to configure the IEEE 802.11 upper layer to buffer broadcast and multicast frames when there are power saving stations so that the driver can fetch them with *ieee80211_get_buffered_bc()*.

IEEE80211_HW_SIGNAL_UNSPEC Hardware can provide signal values but we don't know its units. We expect values between 0 and **max_signal**. If possible please provide dB or dBm instead.

IEEE80211_HW_SIGNAL_DBM Hardware gives signal values in dBm, decibel difference from one milliwatt. This is the preferred method since it is standardized between different devices. **max_signal** does not need to be set.

IEEE80211_HW_NEED_DTIM_BEFORE_ASSOC This device needs to get data from beacon before association (i.e. dtim_period).

IEEE80211_HW_SPECTRUM_MGMT Hardware supports spectrum management defined in 802.11h Measurement, Channel Switch, Quieting, TPC

IEEE80211_HW_AMPDU_AGGREGATION Hardware supports 11n A-MPDU aggregation.

IEEE80211_HW_SUPPORTS_PS Hardware has power save support (i.e. can go to sleep).

IEEE80211_HW_PS_NULLFUNC_STACK Hardware requires nullfunc frame handling in stack, implies stack support for dynamic PS.

IEEE80211_HW_SUPPORTS_DYNAMIC_PS Hardware has support for dynamic PS.

IEEE80211_HW_MFP_CAPABLE Hardware supports management frame protection (MFP, IEEE 802.11w).

IEEE80211_HW_WANT_MONITOR_VIF The driver would like to be informed of a virtual monitor interface when monitor interfaces are the only active interfaces.

IEEE80211_HW_NO_AUTO_VIF The driver would like for no wlanX to be created. It is expected user-space will create vifs as desired (and thus have them named as desired).

IEEE80211_HW_SW_CRYPTO_CONTROL The driver wants to control which of the crypto algorithms can be done in software - so don't automatically try to fall back to it if hardware crypto fails, but do so only if the driver returns 1. This also forces the driver to advertise its supported cipher suites.

IEEE80211_HW_SUPPORT_FAST_XMIT The driver/hardware supports fast-xmit, this currently requires only the ability to calculate the duration for frames.

IEEE80211_HW_REPORTS_TX_ACK_STATUS Hardware can provide ack status reports of Tx frames to the stack.

IEEE80211_HW_CONNECTION_MONITOR The hardware performs its own connection monitoring, including periodic keep-alives to the AP and probing the AP on beacon loss.

IEEE80211_HW_QUEUE_CONTROL The driver wants to control per-interface queue mapping in order to use different queues (not just one per AC) for different virtual interfaces. See the doc section on HW queue control for more details.

IEEE80211_HW_SUPPORTS_PER_STA_GTK The device's crypto engine supports per-station GTKs as used by IBSS RSN or during fast transition. If the device doesn't support per-station GTKs, but can be asked not to decrypt group addressed frames, then IBSS RSN support is still possible but software crypto will be used. Advertise the wiphy flag only in that case.

IEEE80211_HW_AP_LINK_PS When operating in AP mode the device autonomously manages the PS status of connected stations. When this flag is set mac80211 will not trigger PS mode for connected stations based on the PM bit of incoming frames. Use `ieee80211_start_ps()`/`ieee8021_end_ps()` to manually configure the PS mode of connected stations.

IEEE80211_HW_TX_AMPDU_SETUP_IN_HW The device handles TX A-MPDU session setup strictly in HW. mac80211 should not attempt to do this in software.

IEEE80211_HW_SUPPORTS_RC_TABLE The driver supports using a rate selection table provided by the rate control algorithm.

IEEE80211_HW_P2P_DEV_ADDR_FOR_INTF Use the P2P Device address for any P2P Interface. This will be honoured even if more than one interface is supported.

IEEE80211_HW_TIMING_BEACON_ONLY Use sync timing from beacon frames only, to allow getting TBTT of a DTIM beacon.

IEEE80211_HW_SUPPORTS_HT_CCK_RATES Hardware supports mixing HT/CCK rates and can cope with CCK rates in an aggregation session (e.g. by not using aggregation for such frames.)

IEEE80211_HW_CHANCTX_STA_CSA Support 802.11h based channel-switch (CSA) for a single active channel while using channel contexts. When support is not enabled the default action is to disconnect when getting the CSA frame.

IEEE80211_HW_SUPPORTS_CLONED_SKBS The driver will never modify the payload or tailroom of TX skbs without copying them first.

IEEE80211_HW_SINGLE_SCAN_ON_ALL_BANDS The HW supports scanning on all bands in one command, mac80211 doesn't have to run separate scans per band.

IEEE80211_HW_TDLS_WIDER_BW The device/driver supports wider bandwidth than then BSS bandwidth for a TDLS link on the base channel.

IEEE80211_HW_SUPPORTS_AMSDU_IN_AMPDU The driver supports receiving A-MSDUs within A-MPDU.

IEEE80211_HW_BEACON_TX_STATUS The device/driver provides TX status for sent beacons.

IEEE80211_HW_NEEDS_UNIQUE_STA_ADDR Hardware (or driver) requires that each station has a unique address, i.e. each station entry can be identified by just its MAC address; this prevents, for example, the same station from connecting to two virtual AP interfaces at the same time.

IEEE80211_HW_SUPPORTS_REORDERING_BUFFER Hardware (or driver) manages the reordering buffer internally, guaranteeing mac80211 receives frames in order and does not need to manage its own reorder buffer or BA session timeout.

IEEE80211_HW_USES_RSS The device uses RSS and thus requires parallel RX, which implies using per-CPU station statistics.

IEEE80211_HW_TX_AMSDU Hardware (or driver) supports software aggregated A-MSDU frames. Requires software tx queueing and fast-xmit support. When not using minstrel/minstrel_ht rate control, the driver must limit the maximum A-MSDU size based on the current tx rate by setting max_rc_amsdu_len in struct ieee80211_sta.

IEEE80211_HW_TX_FRAG_LIST Hardware (or driver) supports sending frag_list skbs, needed for zero-copy software A-MSDU.

IEEE80211_HW_REPORTS_LOW_ACK The driver (or firmware) reports low ack event by ieee80211_report_low_ack() based on its own algorithm. For such drivers, mac80211 packet loss mechanism will not be triggered and driver is completely depending on firmware event for station kickout.

IEEE80211_HW_SUPPORTS_TX_FRAG Hardware does fragmentation by itself. The stack will not do fragmentation. The callback for **set_frag_threshold** should be set as well.

IEEE80211_HW_SUPPORTS_TDLS_BUFFER_STA Hardware supports buffer STA on TDLS links.

NUM_IEEE80211_HW_FLAGS number of hardware flags, used for sizing arrays

Description

These flags are used to indicate hardware capabilities to the stack. Generally, flags here should have their meaning done in a way that the simplest hardware doesn't need setting any particular flags. There are some exceptions to this rule, however, so you are advised to review these flags carefully.

void **SET_IEEE80211_DEV**(struct *ieee80211_hw* * *hw*, struct *device* * *dev*)
set device for 802.11 hardware

Parameters

struct ieee80211_hw * hw the *struct ieee80211_hw* to set the device for

struct device * dev the *struct device* of this 802.11 device

void **SET_IEEE80211_PERM_ADDR**(struct *ieee80211_hw* * *hw*, const u8 * *addr*)
set the permanent MAC address for 802.11 hardware

Parameters

struct ieee80211_hw * hw the *struct ieee80211_hw* to set the MAC address for

const u8 * addr the address to set

struct **ieee80211_ops**
callbacks from mac80211 to the driver

Definition

```
struct ieee80211_ops {
  void (*tx)(struct ieee80211_hw *hw,struct ieee80211_tx_control *control, struct sk_buff *skb);
  int (*start)(struct ieee80211_hw *hw);
  void (*stop)(struct ieee80211_hw *hw);
#ifdef CONFIG_PM;
  int (*suspend)(struct ieee80211_hw *hw, struct cfg80211_wowlan *wowlan);
  int (*resume)(struct ieee80211_hw *hw);
  void (*set_wakeup)(struct ieee80211_hw *hw, bool enabled);
#endif;
  int (*add_interface)(struct ieee80211_hw *hw, struct ieee80211_vif *vif);
  int (*change_interface)(struct ieee80211_hw *hw,struct ieee80211_vif *vif, enum nl80211_iftype new_typ
  void (*remove_interface)(struct ieee80211_hw *hw, struct ieee80211_vif *vif);
  int (*config)(struct ieee80211_hw *hw, u32 changed);
  void (*bss_info_changed)(struct ieee80211_hw *hw,struct ieee80211_vif *vif,struct ieee80211_bss_conf *
  int (*start_ap)(struct ieee80211_hw *hw, struct ieee80211_vif *vif);
  void (*stop_ap)(struct ieee80211_hw *hw, struct ieee80211_vif *vif);
  u64 (*prepare_multicast)(struct ieee80211_hw *hw, struct netdev_hw_addr_list *mc_list);
  void (*configure_filter)(struct ieee80211_hw *hw,unsigned int changed_flags,unsigned int *total_flags,
  void (*config_iface_filter)(struct ieee80211_hw *hw,struct ieee80211_vif *vif,unsigned int filter_flag
  int (*set_tim)(struct ieee80211_hw *hw, struct ieee80211_sta *sta, bool set);
  int (*set_key)(struct ieee80211_hw *hw, enum set_key_cmd cmd,struct ieee80211_vif *vif, struct ieee802
  void (*update_tkip_key)(struct ieee80211_hw *hw,struct ieee80211_vif *vif,struct ieee80211_key_conf *c
  void (*set_rekey_data)(struct ieee80211_hw *hw,struct ieee80211_vif *vif, struct cfg80211_gtk_rekey_da
  void (*set_default_unicast_key)(struct ieee80211_hw *hw, struct ieee80211_vif *vif, int idx);
  int (*hw_scan)(struct ieee80211_hw *hw, struct ieee80211_vif *vif, struct ieee80211_scan_request *req)
  void (*cancel_hw_scan)(struct ieee80211_hw *hw, struct ieee80211_vif *vif);
  int (*sched_scan_start)(struct ieee80211_hw *hw,struct ieee80211_vif *vif,struct cfg80211_sched_scan_r
  int (*sched_scan_stop)(struct ieee80211_hw *hw, struct ieee80211_vif *vif);
  void (*sw_scan_start)(struct ieee80211_hw *hw,struct ieee80211_vif *vif, const u8 *mac_addr);
  void (*sw_scan_complete)(struct ieee80211_hw *hw, struct ieee80211_vif *vif);
  int (*get_stats)(struct ieee80211_hw *hw, struct ieee80211_low_level_stats *stats);
  void (*get_key_seq)(struct ieee80211_hw *hw,struct ieee80211_key_conf *key, struct ieee80211_key_seq *
  int (*set_frag_threshold)(struct ieee80211_hw *hw, u32 value);
  int (*set_rts_threshold)(struct ieee80211_hw *hw, u32 value);
  int (*sta_add)(struct ieee80211_hw *hw, struct ieee80211_vif *vif, struct ieee80211_sta *sta);
  int (*sta_remove)(struct ieee80211_hw *hw, struct ieee80211_vif *vif, struct ieee80211_sta *sta);
#ifdef CONFIG_MAC80211_DEBUGFS;
  void (*sta_add_debugfs)(struct ieee80211_hw *hw,struct ieee80211_vif *vif,struct ieee80211_sta *sta, s
#endif;
  void (*sta_notify)(struct ieee80211_hw *hw, struct ieee80211_vif *vif, enum sta_notify_cmd, struct iee
  int (*sta_state)(struct ieee80211_hw *hw, struct ieee80211_vif *vif,struct ieee80211_sta *sta,enum iee
  void (*sta_pre_rcu_remove)(struct ieee80211_hw *hw,struct ieee80211_vif *vif, struct ieee80211_sta *st
  void (*sta_rc_update)(struct ieee80211_hw *hw,struct ieee80211_vif *vif,struct ieee80211_sta *sta, u32
  void (*sta_rate_tbl_update)(struct ieee80211_hw *hw,struct ieee80211_vif *vif, struct ieee80211_sta *s
  void (*sta_statistics)(struct ieee80211_hw *hw,struct ieee80211_vif *vif,struct ieee80211_sta *sta, st
  int (*conf_tx)(struct ieee80211_hw *hw,struct ieee80211_vif *vif, u16 ac, const struct ieee80211_tx_qu
  u64 (*get_tsf)(struct ieee80211_hw *hw, struct ieee80211_vif *vif);
  void (*set_tsf)(struct ieee80211_hw *hw, struct ieee80211_vif *vif, u64 tsf);
  void (*offset_tsf)(struct ieee80211_hw *hw, struct ieee80211_vif *vif, s64 offset);
  void (*reset_tsf)(struct ieee80211_hw *hw, struct ieee80211_vif *vif);
  int (*tx_last_beacon)(struct ieee80211_hw *hw);
  int (*ampdu_action)(struct ieee80211_hw *hw,struct ieee80211_vif *vif, struct ieee80211_ampdu_params *
  int (*get_survey)(struct ieee80211_hw *hw, int idx, struct survey_info *survey);
  void (*rfkill_poll)(struct ieee80211_hw *hw);
  void (*set_coverage_class)(struct ieee80211_hw *hw, s16 coverage_class);
#ifdef CONFIG_NL80211_TESTMODE;
  int (*testmode_cmd)(struct ieee80211_hw *hw, struct ieee80211_vif *vif, void *data, int len);
  int (*testmode_dump)(struct ieee80211_hw *hw, struct sk_buff *skb,struct netlink_callback *cb, void *d
#endif;
  void (*flush)(struct ieee80211_hw *hw, struct ieee80211_vif *vif, u32 queues, bool drop);
  void (*channel_switch)(struct ieee80211_hw *hw,struct ieee80211_vif *vif, struct ieee80211_channel_swi
  int (*set_antenna)(struct ieee80211_hw *hw, u32 tx_ant, u32 rx_ant);
  int (*get_antenna)(struct ieee80211_hw *hw, u32 *tx_ant, u32 *rx_ant);
  int (*remain_on_channel)(struct ieee80211_hw *hw,struct ieee80211_vif *vif,struct ieee80211_channel *c
  int (*cancel_remain_on_channel)(struct ieee80211_hw *hw);
  int (*set_ringparam)(struct ieee80211_hw *hw, u32 tx, u32 rx);
  void (*get_ringparam)(struct ieee80211_hw *hw, u32 *tx, u32 *tx_max, u32 *rx, u32 *rx_max);
  bool (*tx_frames_pending)(struct ieee80211_hw *hw);
  int (*set_bitrate_mask)(struct ieee80211_hw *hw, struct ieee80211_vif *vif, const struct cfg80211_bitr
  void (*event_callback)(struct ieee80211_hw *hw,struct ieee80211_vif *vif, const struct ieee80211_event
  void (*allow_buffered_frames)(struct ieee80211_hw *hw,struct ieee80211_sta *sta,u16 tids, int num_fram
  void (*release_buffered_frames)(struct ieee80211_hw *hw,struct ieee80211_sta *sta,u16 tids, int num_fr
  int (*get_et_sset_count)(struct ieee80211_hw *hw, struct ieee80211_vif *vif, int sset);
  void (*get_et_stats)(struct ieee80211_hw *hw,struct ieee80211_vif *vif, struct ethtool_stats *stats, u
  void (*get_et_strings)(struct ieee80211_hw *hw,struct ieee80211_vif *vif, u32 sset, u8 *data);
  void (*mgd_prepare_tx)(struct ieee80211_hw *hw, struct ieee80211_vif *vif);
  void (*mgd_protect_tdls_discover)(struct ieee80211_hw *hw, struct ieee80211_vif *vif);
  int (*add_chanctx)(struct ieee80211_hw *hw, struct ieee80211_chanctx_conf *ctx);
  void (*remove_chanctx)(struct ieee80211_hw *hw, struct ieee80211_chanctx_conf *ctx);
  void (*change_chanctx)(struct ieee80211_hw *hw,struct ieee80211_chanctx_conf *ctx, u32 changed);
  int (*assign_vif_chanctx)(struct ieee80211_hw *hw,struct ieee80211_vif *vif, struct ieee80211_chanctx_
  void (*unassign_vif_chanctx)(struct ieee80211_hw *hw,struct ieee80211_vif *vif, struct ieee80211_chanc
  int (*switch_vif_chanctx)(struct ieee80211_hw *hw,struct ieee80211_vif_chanctx_switch *vifs,int n_vifs
  void (*reconfig_complete)(struct ieee80211_hw *hw, enum ieee80211_reconfig_type reconfig_type);
#if IS_ENABLED(CONFIG_IPV6);
  void (*ipv6_addr_change)(struct ieee80211_hw *hw,struct ieee80211_vif *vif, struct inet6_dev *idev);
#endif;
```

```
  void (*channel_switch_beacon)(struct ieee80211_hw *hw,struct ieee80211_vif *vif, struct cfg80211_chan_
  int (*pre_channel_switch)(struct ieee80211_hw *hw,struct ieee80211_vif *vif, struct ieee80211_channel_
  int (*post_channel_switch)(struct ieee80211_hw *hw, struct ieee80211_vif *vif);
  int (*join_ibss)(struct ieee80211_hw *hw, struct ieee80211_vif *vif);
  void (*leave_ibss)(struct ieee80211_hw *hw, struct ieee80211_vif *vif);
  u32 (*get_expected_throughput)(struct ieee80211_hw *hw, struct ieee80211_sta *sta);
  int (*get_txpower)(struct ieee80211_hw *hw, struct ieee80211_vif *vif, int *dbm);
  int (*tdls_channel_switch)(struct ieee80211_hw *hw,struct ieee80211_vif *vif,struct ieee80211_sta *sta
  void (*tdls_cancel_channel_switch)(struct ieee80211_hw *hw,struct ieee80211_vif *vif, struct ieee80211
  void (*tdls_recv_channel_switch)(struct ieee80211_hw *hw,struct ieee80211_vif *vif, struct ieee80211_t
  void (*wake_tx_queue)(struct ieee80211_hw *hw, struct ieee80211_txq *txq);
  void (*sync_rx_queues)(struct ieee80211_hw *hw);
  int (*start_nan)(struct ieee80211_hw *hw,struct ieee80211_vif *vif, struct cfg80211_nan_conf *conf);
  int (*stop_nan)(struct ieee80211_hw *hw, struct ieee80211_vif *vif);
  int (*nan_change_conf)(struct ieee80211_hw *hw,struct ieee80211_vif *vif, struct cfg80211_nan_conf *co
  int (*add_nan_func)(struct ieee80211_hw *hw,struct ieee80211_vif *vif, const struct cfg80211_nan_func 
  void (*del_nan_func)(struct ieee80211_hw *hw,struct ieee80211_vif *vif, u8 instance_id);
};
```

Members

`tx` Handler that 802.11 module calls for each transmitted frame. skb contains the buffer starting from the IEEE 802.11 header. The low-level driver should send the frame out based on configuration in the TX control data. This handler should, preferably, never fail and stop queues appropriately. Must be atomic.

`start` Called before the first netdevice attached to the hardware is enabled. This should turn on the hardware and must turn on frame reception (for possibly enabled monitor interfaces.) Returns negative error codes, these may be seen in userspace, or zero. When the device is started it should not have a MAC address to avoid acknowledging frames before a non-monitor device is added. Must be implemented and can sleep.

`stop` Called after last netdevice attached to the hardware is disabled. This should turn off the hardware (at least it must turn off frame reception.) May be called right after add_interface if that rejects an interface. If you added any work onto the mac80211 workqueue you should ensure to cancel it on this callback. Must be implemented and can sleep.

`suspend` Suspend the device; mac80211 itself will quiesce before and stop transmitting and doing any other configuration, and then ask the device to suspend. This is only invoked when WoWLAN is configured, otherwise the device is deconfigured completely and reconfigured at resume time. The driver may also impose special conditions under which it wants to use the "normal" suspend (deconfigure), say if it only supports WoWLAN when the device is associated. In this case, it must return 1 from this function.

`resume` If WoWLAN was configured, this indicates that mac80211 is now resuming its operation, after this the device must be fully functional again. If this returns an error, the only way out is to also unregister the device. If it returns 1, then mac80211 will also go through the regular complete restart on resume.

`set_wakeup` Enable or disable wakeup when WoWLAN configuration is modified. The reason is that `device_set_wakeup_enable()` is supposed to be called when the configuration changes, not only in `suspend()`.

`add_interface` Called when a netdevice attached to the hardware is enabled. Because it is not called for monitor mode devices, **start** and **stop** must be implemented. The driver should perform any initialization it needs before the device can be enabled. The initial configuration for the interface is given in the conf parameter. The callback may refuse to add an interface by returning a negative error code (which will be seen in userspace.) Must be implemented and can sleep.

`change_interface` Called when a netdevice changes type. This callback is optional, but only if it is supported can interface types be switched while the interface is UP. The callback may sleep. Note that while an interface is being switched, it will not be found by the interface iteration callbacks.

`remove_interface` Notifies a driver that an interface is going down. The **stop** callback is called after this if it is the last interface and no monitor interfaces are present. When all interfaces are removed, the MAC address in the hardware must be cleared so the device no longer acknowledges packets, the mac_addr member of the conf structure is, however, set to the MAC address of the device going away. Hence, this callback must be implemented. It can sleep.

`config` Handler for configuration requests. IEEE 802.11 code calls this function to change hardware configuration, e.g., channel. This function should never fail but returns a negative error code if it does. The callback can sleep.

`bss_info_changed` Handler for configuration requests related to BSS parameters that may vary during BSS's lifespan, and may affect low level driver (e.g. assoc/disassoc status, erp parameters). This function should not be used if no BSS has been set, unless for association indication. The **changed** parameter indicates which of the bss parameters has changed when a call is made. The callback can sleep.

`start_ap` Start operation on the AP interface, this is called after all the information in bss_conf is set and beacon can be retrieved. A channel context is bound before this is called. Note that if the driver uses software scan or ROC, this (and **stop_ap**) isn't called when the AP is just "paused" for scanning/ROC, which is indicated by the beacon being disabled/enabled via **bss_info_changed**.

`stop_ap` Stop operation on the AP interface.

`prepare_multicast` Prepare for multicast filter configuration. This callback is optional, and its return value is passed to `configure_filter()`. This callback must be atomic.

`configure_filter` Configure the device's RX filter. See the section "Frame filtering" for more information. This callback must be implemented and can sleep.

`config_iface_filter` Configure the interface's RX filter. This callback is optional and is used to configure which frames should be passed to mac80211. The filter_flags is the combination of FIF_* flags. The changed_flags is a bit mask that indicates which flags are changed. This callback can sleep.

`set_tim` Set TIM bit. mac80211 calls this function when a TIM bit must be set or cleared for a given STA. Must be atomic.

`set_key` See the section "Hardware crypto acceleration" This callback is only called between add_interface and remove_interface calls, i.e. while the given virtual interface is enabled. Returns a negative error code if the key can't be added. The callback can sleep.

`update_tkip_key` See the section "Hardware crypto acceleration" This callback will be called in the context of Rx. Called for drivers which set IEEE80211_KEY_FLAG_TKIP_REQ_RX_P1_KEY. The callback must be atomic.

`set_rekey_data` If the device supports GTK rekeying, for example while the host is suspended, it can assign this callback to retrieve the data necessary to do GTK rekeying, this is the KEK, KCK and replay counter. After rekeying was done it should (for example during resume) notify userspace of the new replay counter using `ieee80211_gtk_rekey_notify()`.

`set_default_unicast_key` Set the default (unicast) key index, useful for WEP when the device sends data packets autonomously, e.g. for ARP offloading. The index can be 0-3, or -1 for unsetting it.

`hw_scan` Ask the hardware to service the scan request, no need to start the scan state machine in stack. The scan must honour the channel configuration done by the regulatory agent in the wiphy's registered bands. The hardware (or the driver) needs to make sure that power save is disabled. The **req** ie/ie_len members are rewritten by mac80211 to contain the entire IEs after the SSID, so that drivers need not look at these at all but just send them after the SSID – mac80211 includes the (extended) supported rates and HT information (where applicable). When the scan finishes, *`ieee80211_scan_completed()`* must be called; note that it also must be called when the scan cannot finish due to any error unless this callback returned a negative error code. The callback can sleep.

`cancel_hw_scan` Ask the low-level tp cancel the active hw scan. The driver should ask the hardware to cancel the scan (if possible), but the scan will be completed only after the driver will call *`ieee80211_scan_completed()`*. This callback is needed for wowlan, to prevent enqueueing a new scan_work after the low-level driver was already suspended. The callback can sleep.

sched_scan_start Ask the hardware to start scanning repeatedly at specific intervals. The driver must call the `ieee80211_sched_scan_results()` function whenever it finds results. This process will continue until sched_scan_stop is called.

sched_scan_stop Tell the hardware to stop an ongoing scheduled scan. In this case, `ieee80211_sched_scan_stopped()` must not be called.

sw_scan_start Notifier function that is called just before a software scan is started. Can be NULL, if the driver doesn't need this notification. The mac_addr parameter allows supporting NL80211_SCAN_FLAG_RANDOM_ADDR, the driver may set the NL80211_FEATURE_SCAN_RANDOM_MAC_ADDR flag if it can use this parameter. The callback can sleep.

sw_scan_complete Notifier function that is called just after a software scan finished. Can be NULL, if the driver doesn't need this notification. The callback can sleep.

get_stats Return low-level statistics. Returns zero if statistics are available. The callback can sleep.

get_key_seq If your device implements encryption in hardware and does IV/PN assignment then this callback should be provided to read the IV/PN for the given key from hardware. The callback must be atomic.

set_frag_threshold Configuration of fragmentation threshold. Assign this if the device does fragmentation by itself. Note that to prevent the stack from doing fragmentation IEEE80211_HW_SUPPORTS_TX_FRAG should be set as well. The callback can sleep.

set_rts_threshold Configuration of RTS threshold (if device needs it) The callback can sleep.

sta_add Notifies low level driver about addition of an associated station, AP, IBSS/WDS/mesh peer etc. This callback can sleep.

sta_remove Notifies low level driver about removal of an associated station, AP, IBSS/WDS/mesh peer etc. Note that after the callback returns it isn't safe to use the pointer, not even RCU protected; no RCU grace period is guaranteed between returning here and freeing the station. See **sta_pre_rcu_remove** if needed. This callback can sleep.

sta_add_debugfs Drivers can use this callback to add debugfs files when a station is added to mac80211's station list. This callback should be within a CONFIG_MAC80211_DEBUGFS conditional. This callback can sleep.

sta_notify Notifies low level driver about power state transition of an associated station, AP, IBSS/WDS/mesh peer etc. For a VIF operating in AP mode, this callback will not be called when the flag `IEEE80211_HW_AP_LINK_PS` is set. Must be atomic.

sta_state Notifies low level driver about state transition of a station (which can be the AP, a client, IBSS/WDS/mesh peer etc.) This callback is mutually exclusive with **sta_add**/**sta_remove**. It must not fail for down transitions but may fail for transitions up the list of states. Also note that after the callback returns it isn't safe to use the pointer, not even RCU protected - no RCU grace period is guaranteed between returning here and freeing the station. See **sta_pre_rcu_remove** if needed. The callback can sleep.

sta_pre_rcu_remove Notify driver about station removal before RCU synchronisation. This is useful if a driver needs to have station pointers protected using RCU, it can then use this call to clear the pointers instead of waiting for an RCU grace period to elapse in **sta_state**. The callback can sleep.

sta_rc_update Notifies the driver of changes to the bitrates that can be used to transmit to the station. The changes are advertised with bits from *enum ieee80211_rate_control_changed* and the values are reflected in the station data. This callback should only be used when the driver uses hardware rate control (`IEEE80211_HW_HAS_RATE_CONTROL`) since otherwise the rate control algorithm is notified directly. Must be atomic.

sta_rate_tbl_update Notifies the driver that the rate table changed. This is only used if the configured rate control algorithm actually uses the new rate table API, and is therefore optional. Must be atomic.

sta_statistics Get statistics for this station. For example with beacon filtering, the statistics kept by mac80211 might not be accurate, so let the driver pre-fill the statistics. The driver can fill most of the values (indicating which by setting the filled bitmap), but not all of them make sense - see the source for which ones are possible. Statistics that the driver doesn't fill will be filled by mac80211. The callback can sleep.

`conf_tx` Configure TX queue parameters (EDCF (aifs, cw_min, cw_max), bursting) for a hardware TX queue. Returns a negative error code on failure. The callback can sleep.

`get_tsf` Get the current TSF timer value from firmware/hardware. Currently, this is only used for IBSS mode BSSID merging and debugging. Is not a required function. The callback can sleep.

`set_tsf` Set the TSF timer to the specified value in the firmware/hardware. Currently, this is only used for IBSS mode debugging. Is not a required function. The callback can sleep.

`offset_tsf` Offset the TSF timer by the specified value in the firmware/hardware. Preferred to set_tsf as it avoids delay between calling `set_tsf()` and hardware getting programmed, which will show up as TSF delay. Is not a required function. The callback can sleep.

`reset_tsf` Reset the TSF timer and allow firmware/hardware to synchronize with other STAs in the IBSS. This is only used in IBSS mode. This function is optional if the firmware/hardware takes full care of TSF synchronization. The callback can sleep.

`tx_last_beacon` Determine whether the last IBSS beacon was sent by us. This is needed only for IBSS mode and the result of this function is used to determine whether to reply to Probe Requests. Returns non-zero if this device sent the last beacon. The callback can sleep.

`ampdu_action` Perform a certain A-MPDU action. The RA/TID combination determines the destination and TID we want the ampdu action to be performed for. The action is defined through ieee80211_ampdu_mlme_action. When the action is set to `IEEE80211_AMPDU_TX_OPERATIONAL` the driver may neither send aggregates containing more subframes than **buf_size** nor send aggregates in a way that lost frames would exceed the buffer size. If just limiting the aggregate size, this would be possible with a buf_size of 8:

- `TX: 1.....7`
- `RX: 2....7` (lost frame #1)
- `TX: 8..1...`

which is invalid since #1 was now re-transmitted well past the buffer size of 8. Correct ways to retransmit #1 would be:

- `TX: 1 or`
- `TX: 18 or`
- `TX: 81`

Even 189 would be wrong since 1 could be lost again.

Returns a negative error code on failure. The callback can sleep.

`get_survey` Return per-channel survey information

`rfkill_poll` Poll rfkill hardware state. If you need this, you also need to set wiphy->rfkill_poll to `true` before registration, and need to call *`wiphy_rfkill_set_hw_state()`* in the callback. The callback can sleep.

`set_coverage_class` Set slot time for given coverage class as specified in IEEE 802.11-2007 section 17.3.8.6 and modify ACK timeout accordingly; coverage class equals to -1 to enable ACK timeout estimation algorithm (dynack). To disable dynack set valid value for coverage class. This callback is not required and may sleep.

`testmode_cmd` Implement a cfg80211 test mode command. The passed **vif** may be `NULL`. The callback can sleep.

`testmode_dump` Implement a cfg80211 test mode dump. The callback can sleep.

`flush` Flush all pending frames from the hardware queue, making sure that the hardware queues are empty. The **queues** parameter is a bitmap of queues to flush, which is useful if different virtual interfaces use different hardware queues; it may also indicate all queues. If the parameter **drop** is set to `true`, pending frames may be dropped. Note that vif can be NULL. The callback can sleep.

`channel_switch` Drivers that need (or want) to offload the channel switch operation for CSAs received from the AP may implement this callback. They must then call `ieee80211_chswitch_done()` to indicate completion of the channel switch.

`set_antenna` Set antenna configuration (tx_ant, rx_ant) on the device. Parameters are bitmaps of allowed antennas to use for TX/RX. Drivers may reject TX/RX mask combinations they cannot support by returning -EINVAL (also see nl80211.h **NL80211_ATTR_WIPHY_ANTENNA_TX**).

`get_antenna` Get current antenna configuration from device (tx_ant, rx_ant).

`remain_on_channel` Starts an off-channel period on the given channel, must call back to `ieee80211_ready_on_channel()` when on that channel. Note that normal channel traffic is not stopped as this is intended for hw offload. Frames to transmit on the off-channel channel are transmitted normally except for the `IEEE80211_TX_CTL_TX_OFFCHAN` flag. When the duration (which will always be non-zero) expires, the driver must call `ieee80211_remain_on_channel_expired()`. Note that this callback may be called while the device is in IDLE and must be accepted in this case. This callback may sleep.

`cancel_remain_on_channel` Requests that an ongoing off-channel period is aborted before it expires. This callback may sleep.

`set_ringparam` Set tx and rx ring sizes.

`get_ringparam` Get tx and rx ring current and maximum sizes.

`tx_frames_pending` Check if there is any pending frame in the hardware queues before entering power save.

`set_bitrate_mask` Set a mask of rates to be used for rate control selection when transmitting a frame. Currently only legacy rates are handled. The callback can sleep.

`event_callback` Notify driver about any event in mac80211. See `enum ieee80211_event_type` for the different types. The callback must be atomic.

`allow_buffered_frames` Prepare device to allow the given number of frames to go out to the given station. The frames will be sent by mac80211 via the usual TX path after this call. The TX information for frames released will also have the `IEEE80211_TX_CTL_NO_PS_BUFFER` flag set and the last one will also have `IEEE80211_TX_STATUS_EOSP` set. In case frames from multiple TIDs are released and the driver might reorder them between the TIDs, it must set the `IEEE80211_TX_STATUS_EOSP` flag on the last frame and clear it on all others and also handle the EOSP bit in the QoS header correctly. Alternatively, it can also call the *`ieee80211_sta_eosp()`* function. The **tids** parameter is a bitmap and tells the driver which TIDs the frames will be on; it will at most have two bits set. This callback must be atomic.

`release_buffered_frames` Release buffered frames according to the given parameters. In the case where the driver buffers some frames for sleeping stations mac80211 will use this callback to tell the driver to release some frames, either for PS-poll or uAPSD. Note that if the **more_data** parameter is `false` the driver must check if there are more frames on the given TIDs, and if there are more than the frames being released then it must still set the more-data bit in the frame. If the **more_data** parameter is `true`, then of course the more-data bit must always be set. The **tids** parameter tells the driver which TIDs to release frames from, for PS-poll it will always have only a single bit set. In the case this is used for a PS-poll initiated release, the **num_frames** parameter will always be 1 so code can be shared. In this case the driver must also set `IEEE80211_TX_STATUS_EOSP` flag on the TX status (and must report TX status) so that the PS-poll period is properly ended. This is used to avoid sending multiple responses for a retried PS-poll frame. In the case this is used for uAPSD, the **num_frames** parameter may be bigger than one, but the driver may send fewer frames (it must send at least one, however). In this case it is also responsible for setting the EOSP flag in the QoS header of the frames. Also, when the service period ends, the driver must set `IEEE80211_TX_STATUS_EOSP` on the last frame in the SP. Alternatively, it may call the function *`ieee80211_sta_eosp()`* to inform mac80211 of the end of the SP. This callback must be atomic.

`get_et_sset_count` Ethtool API to get string-set count.

get_et_stats Ethtool API to get a set of u64 stats.

get_et_strings Ethtool API to get a set of strings to describe stats and perhaps other supported types of ethtool data-sets.

mgd_prepare_tx Prepare for transmitting a management frame for association before associated. In multi-channel scenarios, a virtual interface is bound to a channel before it is associated, but as it isn't associated yet it need not necessarily be given airtime, in particular since any transmission to a P2P GO needs to be synchronized against the GO's powersave state. mac80211 will call this function before transmitting a management frame prior to having successfully associated to allow the driver to give it channel time for the transmission, to get a response and to be able to synchronize with the GO. The callback will be called before each transmission and upon return mac80211 will transmit the frame right away. The callback is optional and can (should!) sleep.

mgd_protect_tdls_discover Protect a TDLS discovery session. After sending a TDLS discovery-request, we expect a reply to arrive on the AP's channel. We must stay on the channel (no PSM, scan, etc.), since a TDLS setup-response is a direct packet not buffered by the AP. mac80211 will call this function just before the transmission of a TDLS discovery-request. The recommended period of protection is at least 2 * (DTIM period). The callback is optional and can sleep.

add_chanctx Notifies device driver about new channel context creation. This callback may sleep.

remove_chanctx Notifies device driver about channel context destruction. This callback may sleep.

change_chanctx Notifies device driver about channel context changes that may happen when combining different virtual interfaces on the same channel context with different settings This callback may sleep.

assign_vif_chanctx Notifies device driver about channel context being bound to vif. Possible use is for hw queue remapping. This callback may sleep.

unassign_vif_chanctx Notifies device driver about channel context being unbound from vif. This callback may sleep.

switch_vif_chanctx switch a number of vifs from one chanctx to another, as specified in the list of **ieee80211_vif_chanctx_switch** passed to the driver, according to the mode defined in `ieee80211_chanctx_switch_mode`. This callback may sleep.

reconfig_complete Called after a call to `ieee80211_restart_hw()` and during resume, when the reconfiguration has completed. This can help the driver implement the reconfiguration step (and indicate mac80211 is ready to receive frames). This callback may sleep.

ipv6_addr_change IPv6 address assignment on the given interface changed. Currently, this is only called for managed or P2P client interfaces. This callback is optional; it must not sleep.

channel_switch_beacon Starts a channel switch to a new channel. Beacons are modified to include CSA or ECSA IEs before calling this function. The corresponding count fields in these IEs must be decremented, and when they reach 1 the driver must call `ieee80211_csa_finish()`. Drivers which use *`ieee80211_beacon_get()`* get the csa counter decremented by mac80211, but must check if it is 1 using `ieee80211_csa_is_complete()` after the beacon has been transmitted and then call `ieee80211_csa_finish()`. If the CSA count starts as zero or 1, this function will not be called, since there won't be any time to beacon before the switch anyway.

pre_channel_switch This is an optional callback that is called before a channel switch procedure is started (ie. when a STA gets a CSA or a userspace initiated channel-switch), allowing the driver to prepare for the channel switch.

post_channel_switch This is an optional callback that is called after a channel switch procedure is completed, allowing the driver to go back to a normal configuration.

join_ibss Join an IBSS (on an IBSS interface); this is called after all information in bss_conf is set up and the beacon can be retrieved. A channel context is bound before this is called.

leave_ibss Leave the IBSS again.

get_expected_throughput extract the expected throughput towards the specified station. The returned value is expressed in Kbps. It returns 0 if the RC algorithm does not have proper data to provide.

get_txpower get current maximum tx power (in dBm) based on configuration and hardware limits.

tdls_channel_switch Start channel-switching with a TDLS peer. The driver is responsible for continually initiating channel-switching operations and returning to the base channel for communication with the AP. The driver receives a channel-switch request template and the location of the switch-timing IE within the template as part of the invocation. The template is valid only within the call, and the driver can optionally copy the skb for further re-use.

tdls_cancel_channel_switch Stop channel-switching with a TDLS peer. Both peers must be on the base channel when the call completes.

tdls_recv_channel_switch a TDLS channel-switch related frame (request or response) has been received from a remote peer. The driver gets parameters parsed from the incoming frame and may use them to continue an ongoing channel-switch operation. In addition, a channel-switch response template is provided, together with the location of the switch-timing IE within the template. The skb can only be used within the function call.

wake_tx_queue Called when new packets have been added to the queue.

sync_rx_queues Process all pending frames in RSS queues. This is a synchronization which is needed in case driver has in its RSS queues pending frames that were received prior to the control path action currently taken (e.g. disassociation) but are not processed yet.

start_nan join an existing NAN cluster, or create a new one.

stop_nan leave the NAN cluster.

nan_change_conf change NAN configuration. The data in cfg80211_nan_conf contains full new configuration and changes specify which parameters are changed with respect to the last NAN config. The driver gets both full configuration and the changed parameters since some devices may need the full configuration while others need only the changed parameters.

add_nan_func Add a NAN function. Returns 0 on success. The data in cfg80211_nan_func must not be referenced outside the scope of this call.

del_nan_func Remove a NAN function. The driver must call `ieee80211_nan_func_terminated()` with NL80211_NAN_FUNC_TERM_REASON_USER_REQUEST reason code upon removal.

Description

This structure contains various callbacks that the driver may handle or, in some cases, must handle, for example to configure the hardware to a new channel or to transmit a frame.

struct *ieee80211_hw* * **ieee80211_alloc_hw**(size_t *priv_data_len*, const struct *ieee80211_ops* * *ops*)

Allocate a new hardware device

Parameters

size_t priv_data_len length of private data

const struct ieee80211_ops * ops callbacks for this device

Description

This must be called once for each hardware device. The returned pointer must be used to refer to this device when calling other functions. mac80211 allocates a private data area for the driver pointed to by **priv** in *struct ieee80211_hw*, the size of this area is given as **priv_data_len**.

Return

A pointer to the new hardware device, or NULL on error.

int **ieee80211_register_hw**(struct *ieee80211_hw* * *hw*)

Register hardware device

Parameters

struct ieee80211_hw * hw the device to register as returned by *ieee80211_alloc_hw()*

Description

You must call this function before any other functions in mac80211. Note that before a hardware can be registered, you need to fill the contained wiphy's information.

Return

0 on success. An error code otherwise.

void **ieee80211_unregister_hw**(struct *ieee80211_hw* * *hw*)
Unregister a hardware device

Parameters

struct ieee80211_hw * hw the hardware to unregister

Description

This function instructs mac80211 to free allocated resources and unregister netdevices from the networking subsystem.

void **ieee80211_free_hw**(struct *ieee80211_hw* * *hw*)
free hardware descriptor

Parameters

struct ieee80211_hw * hw the hardware to free

Description

This function frees everything that was allocated, including the private data for the driver. You must call *ieee80211_unregister_hw()* before calling this function.

PHY configuration

TBD

This chapter should describe PHY handling including start/stop callbacks and the various structures used.

struct **ieee80211_conf**
configuration of the device

Definition

```
struct ieee80211_conf {
  u32 flags;
  int power_level, dynamic_ps_timeout;
  u16 listen_interval;
  u8 ps_dtim_period;
  u8 long_frame_max_tx_count, short_frame_max_tx_count;
  struct cfg80211_chan_def chandef;
  bool radar_enabled;
  enum ieee80211_smps_mode smps_mode;
};
```

Members

flags configuration flags defined above

power_level requested transmit power (in dBm), backward compatibility value only that is set to the minimum of all interfaces

dynamic_ps_timeout The dynamic powersave timeout (in ms), see the powersave documentation below. This variable is valid only when the CONF_PS flag is set.

listen_interval listen interval in units of beacon interval

ps_dtim_period The DTIM period of the AP we're connected to, for use in power saving. Power saving will not be enabled until a beacon has been received and the DTIM period is known.

long_frame_max_tx_count Maximum number of transmissions for a "long" frame (a frame not RTS protected), called "dot11LongRetryLimit" in 802.11, but actually means the number of transmissions not the number of retries

short_frame_max_tx_count Maximum number of transmissions for a "short" frame, called "dot11ShortRetryLimit" in 802.11, but actually means the number of transmissions not the number of retries

chandef the channel definition to tune to

radar_enabled whether radar detection is enabled

smps_mode spatial multiplexing powersave mode; note that `IEEE80211_SMPS_STATIC` is used when the device is not configured for an HT channel. Note that this is only valid if channel contexts are not used, otherwise each channel context has the number of chains listed.

Description

This struct indicates how the driver shall configure the hardware.

enum **ieee80211_conf_flags**
configuration flags

Constants

IEEE80211_CONF_MONITOR there's a monitor interface present – use this to determine for example whether to calculate timestamps for packets or not, do not use instead of filter flags!

IEEE80211_CONF_PS Enable 802.11 power save mode (managed mode only). This is the power save mode defined by IEEE 802.11-2007 section 11.2, meaning that the hardware still wakes up for beacons, is able to transmit frames and receive the possible acknowledgment frames. Not to be confused with hardware specific wakeup/sleep states, driver is responsible for that. See the section "Powersave support" for more.

IEEE80211_CONF_IDLE The device is running, but idle; if the flag is set the driver should be prepared to handle configuration requests but may turn the device off as much as possible. Typically, this flag will be set when an interface is set UP but not associated or scanning, but it can also be unset in that case when monitor interfaces are active.

IEEE80211_CONF_OFFCHANNEL The device is currently not on its main operating channel.

Description

Flags to define PHY configuration options

Virtual interfaces

TBD

This chapter should describe virtual interface basics that are relevant to the driver (VLANs, MGMT etc are not.) It should explain the use of the add_iface/remove_iface callbacks as well as the interface configuration callbacks.

Things related to AP mode should be discussed there.

Things related to supporting multiple interfaces should be in the appropriate chapter, a BIG FAT note should be here about this though and the recommendation to allow only a single interface in STA mode at first!

struct **ieee80211_vif**
per-interface data

Definition

```
struct ieee80211_vif {
  enum nl80211_iftype type;
  struct ieee80211_bss_conf bss_conf;
  u8 addr[ETH_ALEN] ;
  bool p2p;
  bool csa_active;
  bool mu_mimo_owner;
  u8 cab_queue;
  u8 hw_queue[IEEE80211_NUM_ACS];
  struct ieee80211_txq *txq;
  struct ieee80211_chanctx_conf __rcu *chanctx_conf;
  u32 driver_flags;
#ifdef CONFIG_MAC80211_DEBUGFS;
  struct dentry *debugfs_dir;
#endif;
  unsigned int probe_req_reg;
  u8 drv_priv[0] ;
};
```

Members

type type of this virtual interface

bss_conf BSS configuration for this interface, either our own or the BSS we're associated to

addr address of this interface

p2p indicates whether this AP or STA interface is a p2p interface, i.e. a GO or p2p-sta respectively

csa_active marks whether a channel switch is going on. Internally it is write-protected by sdata_lock and local->mtx so holding either is fine for read access.

mu_mimo_owner indicates interface owns MU-MIMO capability

cab_queue content-after-beacon (DTIM beacon really) queue, AP mode only

hw_queue hardware queue for each AC

txq the multicast data TX queue (if driver uses the TXQ abstraction)

chanctx_conf The channel context this interface is assigned to, or NULL when it is not assigned. This pointer is RCU-protected due to the TX path needing to access it; even though the netdev carrier will always be off when it is NULL there can still be races and packets could be processed after it switches back to NULL.

driver_flags flags/capabilities the driver has for this interface, these need to be set (or cleared) when the interface is added or, if supported by the driver, the interface type is changed at runtime, mac80211 will never touch this field

debugfs_dir debugfs dentry, can be used by drivers to create own per interface debug files. Note that it will be NULL for the virtual monitor interface (if that is requested.)

probe_req_reg probe requests should be reported to mac80211 for this interface.

drv_priv data area for driver use, will always be aligned to sizeof(void *).

Description

Data in this structure is continually present for driver use during the life of a virtual interface.

Receive and transmit processing

what should be here

TBD

This should describe the receive and transmit paths in mac80211/the drivers as well as transmit status handling.

Frame format

As a general rule, when frames are passed between mac80211 and the driver, they start with the IEEE 802.11 header and include the same octets that are sent over the air except for the FCS which should be calculated by the hardware.

There are, however, various exceptions to this rule for advanced features:

The first exception is for hardware encryption and decryption offload where the IV/ICV may or may not be generated in hardware.

Secondly, when the hardware handles fragmentation, the frame handed to the driver from mac80211 is the MSDU, not the MPDU.

Packet alignment

Drivers always need to pass packets that are aligned to two-byte boundaries to the stack.

Additionally, should, if possible, align the payload data in a way that guarantees that the contained IP header is aligned to a four-byte boundary. In the case of regular frames, this simply means aligning the payload to a four-byte boundary (because either the IP header is directly contained, or IV/RFC1042 headers that have a length divisible by four are in front of it). If the payload data is not properly aligned and the architecture doesn't support efficient unaligned operations, mac80211 will align the data.

With A-MSDU frames, however, the payload data address must yield two modulo four because there are 14-byte 802.3 headers within the A-MSDU frames that push the IP header further back to a multiple of four again. Thankfully, the specs were sane enough this time around to require padding each A-MSDU subframe to a length that is a multiple of four.

Padding like Atheros hardware adds which is between the 802.11 header and the payload is not supported, the driver is required to move the 802.11 header to be directly in front of the payload in that case.

Calling into mac80211 from interrupts

Only *ieee80211_tx_status_irqsafe()* and *ieee80211_rx_irqsafe()* can be called in hardware interrupt context. The low-level driver must not call any other functions in hardware interrupt context. If there is a need for such call, the low-level driver should first ACK the interrupt and perform the IEEE 802.11 code call after this, e.g. from a scheduled workqueue or even tasklet function.

NOTE: If the driver opts to use the `_irqsafe()` functions, it may not also use the non-IRQ-safe functions!

functions/definitions

struct **ieee80211_rx_status**
receive status

Definition

```
struct ieee80211_rx_status {
  u64 mactime;
  u64 boottime_ns;
  u32 device_timestamp;
  u32 ampdu_reference;
  u32 flag;
  u16 freq;
  u8 enc_flags;
  u8 encoding:2, bw:3;
  u8 rate_idx;
  u8 nss;
  u8 rx_flags;
  u8 band;
  u8 antenna;
  s8 signal;
  u8 chains;
  s8 chain_signal[IEEE80211_MAX_CHAINS];
  u8 ampdu_delimiter_crc;
};
```

Members

mactime value in microseconds of the 64-bit Time Synchronization Function (TSF) timer when the first data symbol (MPDU) arrived at the hardware.

boottime_ns CLOCK_BOOTTIME timestamp the frame was received at, this is needed only for beacons and probe responses that update the scan cache.

device_timestamp arbitrary timestamp for the device, mac80211 doesn't use it but can store it and pass it back to the driver for synchronisation

ampdu_reference A-MPDU reference number, must be a different value for each A-MPDU but the same for each subframe within one A-MPDU

flag RX_FLAG_*

freq frequency the radio was tuned to when receiving this frame, in MHz This field must be set for management frames, but isn't strictly needed for data (other) frames - for those it only affects radiotap reporting.

enc_flags uses bits from enum `mac80211_rx_encoding_flags`

encoding enum `mac80211_rx_encoding`

bw enum `rate_info_bw`

rate_idx index of data rate into band's supported rates or MCS index if HT or VHT is used (RX_FLAG_HT/RX_FLAG_VHT)

nss number of streams (VHT and HE only)

rx_flags internal RX flags for mac80211

band the active band when this frame was received

antenna antenna used

signal signal strength when receiving this frame, either in dBm, in dB or unspecified depending on the hardware capabilities flags **IEEE80211_HW_SIGNAL_***

chains bitmask of receive chains for which separate signal strength values were filled.

chain_signal per-chain signal strength, in dBm (unlike **signal**, doesn't support dB or unspecified units)

ampdu_delimiter_crc A-MPDU delimiter CRC

Description

The low-level driver should provide this information (the subset supported by hardware) to the 802.11 code with each received frame, in the skb's control buffer (cb).

enum **mac80211_rx_flags**
receive flags

Constants

RX_FLAG_MMIC_ERROR Michael MIC error was reported on this frame. Use together with `RX_FLAG_MMIC_STRIPPED`.

RX_FLAG_DECRYPTED This frame was decrypted in hardware.

RX_FLAG_MACTIME_PLCP_START The timestamp passed in the RX status (**mactime** field) is valid and contains the time the SYNC preamble was received.

RX_FLAG_MMIC_STRIPPED the Michael MIC is stripped off this frame, verification has been done by the hardware.

RX_FLAG_IV_STRIPPED The IV and ICV are stripped from this frame. If this flag is set, the stack cannot do any replay detection hence the driver or hardware will have to do that.

RX_FLAG_FAILED_FCS_CRC Set this flag if the FCS check failed on the frame.

RX_FLAG_FAILED_PLCP_CRC Set this flag if the PCLP check failed on the frame.

RX_FLAG_MACTIME_START The timestamp passed in the RX status (**mactime** field) is valid and contains the time the first symbol of the MPDU was received. This is useful in monitor mode and for proper IBSS merging.

RX_FLAG_NO_SIGNAL_VAL The signal strength value is not present. Valid only for data frames (mainly A-MPDU)

RX_FLAG_AMPDU_DETAILS A-MPDU details are known, in particular the reference number (**ampdu_reference**) must be populated and be a distinct number for each A-MPDU

RX_FLAG_PN_VALIDATED Currently only valid for CCMP/GCMP frames, this flag indicates that the PN was verified for replay protection. Note that this flag is also currently only supported when a frame is also decrypted (ie. **RX_FLAG_DECRYPTED** must be set)

RX_FLAG_DUP_VALIDATED The driver should set this flag if it did de-duplication by itself.

RX_FLAG_AMPDU_LAST_KNOWN last subframe is known, should be set on all subframes of a single A-MPDU

RX_FLAG_AMPDU_IS_LAST this subframe is the last subframe of the A-MPDU

RX_FLAG_AMPDU_DELIM_CRC_ERROR A delimiter CRC error has been detected on this subframe

RX_FLAG_AMPDU_DELIM_CRC_KNOWN The delimiter CRC field is known (the CRC is stored in the **ampdu_delimiter_crc** field)

RX_FLAG_MACTIME_END The timestamp passed in the RX status (**mactime** field) is valid and contains the time the last symbol of the MPDU (including FCS) was received.

RX_FLAG_ONLY_MONITOR Report frame only to monitor interfaces without processing it in any regular way. This is useful if drivers offload some frames but still want to report them for sniffing purposes.

RX_FLAG_SKIP_MONITOR Process and report frame to all interfaces except monitor interfaces. This is useful if drivers offload some frames but still want to report them for sniffing purposes.

RX_FLAG_AMSDU_MORE Some drivers may prefer to report separate A-MSDU subframes instead of a one huge frame for performance reasons. All, but the last MSDU from an A-MSDU should have this flag set. E.g. if an A-MSDU has 3 frames, the first 2 must have the flag set, while the 3rd (last) one must not have this flag set. The flag is used to deal with retransmission/duplication recovery properly since A-MSDU subframes share the same sequence number. Reported subframes can be either regular MSDU or singly A-MSDUs. Subframes must not be interleaved with other frames.

RX_FLAG_RADIOTAP_VENDOR_DATA This frame contains vendor-specific radiotap data in the skb->data (before the frame) as described by the `struct ieee80211_vendor_radiotap`.

RX_FLAG_MIC_STRIPPED The mic was stripped of this packet. Decryption was done by the hardware

RX_FLAG_ALLOW_SAME_PN Allow the same PN as same packet before. This is used for AMSDU subframes which can have the same PN as the first subframe.

RX_FLAG_ICV_STRIPPED The ICV is stripped from this frame. CRC checking must be done in the hardware.

Description

These flags are used with the **flag** member of *struct ieee80211_rx_status*.

enum **`mac80211_tx_info_flags`**
flags to describe transmission information/status

Constants

`IEEE80211_TX_CTL_REQ_TX_STATUS` require TX status callback for this frame.

`IEEE80211_TX_CTL_ASSIGN_SEQ` The driver has to assign a sequence number to this frame, taking care of not overwriting the fragment number and increasing the sequence number only when the IEEE80211_TX_CTL_FIRST_FRAGMENT flag is set. mac80211 will properly assign sequence numbers to QoS-data frames but cannot do so correctly for non-QoS-data and management frames because beacons need them from that counter as well and mac80211 cannot guarantee proper sequencing. If this flag is set, the driver should instruct the hardware to assign a sequence number to the frame or assign one itself. Cf. IEEE 802.11-2007 7.1.3.4.1 paragraph 3. This flag will always be set for beacons and always be clear for frames without a sequence number field.

`IEEE80211_TX_CTL_NO_ACK` tell the low level not to wait for an ack

`IEEE80211_TX_CTL_CLEAR_PS_FILT` clear powersave filter for destination station

`IEEE80211_TX_CTL_FIRST_FRAGMENT` this is a first fragment of the frame

`IEEE80211_TX_CTL_SEND_AFTER_DTIM` send this frame after DTIM beacon

`IEEE80211_TX_CTL_AMPDU` this frame should be sent as part of an A-MPDU

`IEEE80211_TX_CTL_INJECTED` Frame was injected, internal to mac80211.

`IEEE80211_TX_STAT_TX_FILTERED` The frame was not transmitted because the destination STA was in powersave mode. Note that to avoid race conditions, the filter must be set by the hardware or firmware upon receiving a frame that indicates that the station went to sleep (must be done on device to filter frames already on the queue) and may only be unset after mac80211 gives the OK for that by setting the IEEE80211_TX_CTL_CLEAR_PS_FILT (see above), since only then is it guaranteed that no more frames are in the hardware queue.

`IEEE80211_TX_STAT_ACK` Frame was acknowledged

`IEEE80211_TX_STAT_AMPDU` The frame was aggregated, so status is for the whole aggregation.

`IEEE80211_TX_STAT_AMPDU_NO_BACK` no block ack was returned, so consider using block ack request (BAR).

`IEEE80211_TX_CTL_RATE_CTRL_PROBE` internal to mac80211, can be set by rate control algorithms to indicate probe rate, will be cleared for fragmented frames (except on the last fragment)

`IEEE80211_TX_INTFL_OFFCHAN_TX_OK` Internal to mac80211. Used to indicate that a frame can be transmitted while the queues are stopped for off-channel operation.

`IEEE80211_TX_INTFL_NEED_TXPROCESSING` completely internal to mac80211, used to indicate that a pending frame requires TX processing before it can be sent out.

`IEEE80211_TX_INTFL_RETRIED` completely internal to mac80211, used to indicate that a frame was already retried due to PS

`IEEE80211_TX_INTFL_DONT_ENCRYPT` completely internal to mac80211, used to indicate frame should not be encrypted

`IEEE80211_TX_CTL_NO_PS_BUFFER` This frame is a response to a poll frame (PS-Poll or uAPSD) or a non-bufferable MMPDU and must be sent although the station is in powersave mode.

`IEEE80211_TX_CTL_MORE_FRAMES` More frames will be passed to the transmit function after the current frame, this can be used by drivers to kick the DMA queue only if unset or when the queue gets full.

`IEEE80211_TX_INTFL_RETRANSMISSION` This frame is being retransmitted after TX status because the destination was asleep, it must not be modified again (no seqno assignment, crypto, etc.)

`IEEE80211_TX_INTFL_MLME_CONN_TX` This frame was transmitted by the MLME code for connection establishment, this indicates that its status should kick the MLME state machine.

IEEE80211_TX_INTFL_NL80211_FRAME_TX Frame was requested through nl80211 MLME command (internal to mac80211 to figure out whether to send TX status to user space)

IEEE80211_TX_CTL_LDPC tells the driver to use LDPC for this frame

IEEE80211_TX_CTL_STBC Enables Space-Time Block Coding (STBC) for this frame and selects the maximum number of streams that it can use.

IEEE80211_TX_CTL_TX_OFFCHAN Marks this packet to be transmitted on the off-channel channel when a remain-on-channel offload is done in hardware – normal packets still flow and are expected to be handled properly by the device.

IEEE80211_TX_INTFL_TKIP_MIC_FAILURE Marks this packet to be used for TKIP testing. It will be sent out with incorrect Michael MIC key to allow TKIP countermeasures to be tested.

IEEE80211_TX_CTL_NO_CCK_RATE This frame will be sent at non CCK rate. This flag is actually used for management frame especially for P2P frames not being sent at CCK rate in 2GHz band.

IEEE80211_TX_STATUS_EOSP This packet marks the end of service period, when its status is reported the service period ends. For frames in an SP that mac80211 transmits, it is already set; for driver frames the driver may set this flag. It is also used to do the same for PS-Poll responses.

IEEE80211_TX_CTL_USE_MINRATE This frame will be sent at lowest rate. This flag is used to send nullfunc frame at minimum rate when the nullfunc is used for connection monitoring purpose.

IEEE80211_TX_CTL_DONTFRAG Don't fragment this packet even if it would be fragmented by size (this is optional, only used for monitor injection).

IEEE80211_TX_STAT_NOACK_TRANSMITTED A frame that was marked with IEEE80211_TX_CTL_NO_ACK has been successfully transmitted without any errors (like issues specific to the driver/HW). This flag must not be set for frames that don't request no-ack behaviour with IEEE80211_TX_CTL_NO_ACK.

Description

These flags are used with the **flags** member of *ieee80211_tx_info*.

Note

If you have to add new flags to the enumeration, then don't forget to update `IEEE80211_TX_TEMPORARY_FLAGS` when necessary.

enum **mac80211_tx_control_flags**
flags to describe transmit control

Constants

IEEE80211_TX_CTRL_PORT_CTRL_PROTO this frame is a port control protocol frame (e.g. EAP)

IEEE80211_TX_CTRL_PS_RESPONSE This frame is a response to a poll frame (PS-Poll or uAPSD).

IEEE80211_TX_CTRL_RATE_INJECT This frame is injected with rate information

IEEE80211_TX_CTRL_AMSDU This frame is an A-MSDU frame

IEEE80211_TX_CTRL_FAST_XMIT This frame is going through the fast_xmit path

Description

These flags are used in tx_info->control.flags.

enum **mac80211_rate_control_flags**
per-rate flags set by the Rate Control algorithm.

Constants

IEEE80211_TX_RC_USE_RTS_CTS Use RTS/CTS exchange for this rate.

IEEE80211_TX_RC_USE_CTS_PROTECT CTS-to-self protection is required. This is set if the current BSS requires ERP protection.

IEEE80211_TX_RC_USE_SHORT_PREAMBLE Use short preamble.

IEEE80211_TX_RC_MCS HT rate.

IEEE80211_TX_RC_GREEN_FIELD Indicates whether this rate should be used in Greenfield mode.

IEEE80211_TX_RC_40_MHZ_WIDTH Indicates if the Channel Width should be 40 MHz.

IEEE80211_TX_RC_DUP_DATA The frame should be transmitted on both of the adjacent 20 MHz channels, if the current channel type is NL80211_CHAN_HT40MINUS or NL80211_CHAN_HT40PLUS.

IEEE80211_TX_RC_SHORT_GI Short Guard interval should be used for this rate.

IEEE80211_TX_RC_VHT_MCS VHT MCS rate, in this case the idx field is split into a higher 4 bits (Nss) and lower 4 bits (MCS number)

IEEE80211_TX_RC_80_MHZ_WIDTH Indicates 80 MHz transmission

IEEE80211_TX_RC_160_MHZ_WIDTH Indicates 160 MHz transmission (80+80 isn't supported yet)

Description

These flags are set by the Rate control algorithm for each rate during tx, in the **flags** member of struct ieee80211_tx_rate.

struct **ieee80211_tx_rate**
rate selection/status

Definition

```
struct ieee80211_tx_rate {
  s8 idx;
  u16 count:5, flags:11;
};
```

Members

idx rate index to attempt to send with

count number of tries in this rate before going to the next rate

flags rate control flags (*enum mac80211_rate_control_flags*)

Description

A value of -1 for **idx** indicates an invalid rate and, if used in an array of retry rates, that no more rates should be tried.

When used for transmit status reporting, the driver should always report the rate along with the flags it used.

struct ieee80211_tx_info contains an array of these structs in the control information, and it will be filled by the rate control algorithm according to what should be sent. For example, if this array contains, in the format { <idx>, <count> } the information:

```
{ 3, 2 }, { 2, 2 }, { 1, 4 }, { -1, 0 }, { -1, 0 }
```

then this means that the frame should be transmitted up to twice at rate 3, up to twice at rate 2, and up to four times at rate 1 if it doesn't get acknowledged. Say it gets acknowledged by the peer after the fifth attempt, the status information should then contain:

```
{ 3, 2 }, { 2, 2 }, { 1, 1 }, { -1, 0 } ...
```

since it was transmitted twice at rate 3, twice at rate 2 and once at rate 1 after which we received an acknowledgement.

struct **ieee80211_tx_info**
skb transmit information

Definition

```
struct ieee80211_tx_info {
  u32 flags;
  u8 band;
  u8 hw_queue;
  u16 ack_frame_id;
  union {
    struct {
      union {
        struct {
          struct ieee80211_tx_rate rates[ IEEE80211_TX_MAX_RATES];
          s8 rts_cts_rate_idx;
          u8 use_rts:1;
          u8 use_cts_prot:1;
          u8 short_preamble:1;
          u8 skip_table:1;
        };
        unsigned long jiffies;
      };
      struct ieee80211_vif *vif;
      struct ieee80211_key_conf *hw_key;
      u32 flags;
      codel_time_t enqueue_time;
    } control;
    struct {
      u64 cookie;
    } ack;
    struct {
      struct ieee80211_tx_rate rates[IEEE80211_TX_MAX_RATES];
      s32 ack_signal;
      u8 ampdu_ack_len;
      u8 ampdu_len;
      u8 antenna;
      u16 tx_time;
      void *status_driver_data[19 / sizeof(void *)];
    } status;
    struct {
      struct ieee80211_tx_rate driver_rates[ IEEE80211_TX_MAX_RATES];
      u8 pad[4];
      void *rate_driver_data[ IEEE80211_TX_INFO_RATE_DRIVER_DATA_SIZE / sizeof(void *)];
    };
    void *driver_data[ IEEE80211_TX_INFO_DRIVER_DATA_SIZE / sizeof(void *)];
  };
};
```

Members

`flags` transmit info flags, defined above

`band` the band to transmit on (use for checking for races)

`hw_queue` HW queue to put the frame on, `skb_get_queue_mapping()` gives the AC

`ack_frame_id` internal frame ID for TX status, used internally

`{unnamed_union}` anonymous

`control` union for control data

`{unnamed_union}` anonymous

`{unnamed_struct}` anonymous

`status` union for status data

`{unnamed_struct}` anonymous

`driver_data` array of driver_data pointers

Description

This structure is placed in skb->cb for three uses:

1. mac80211 TX control - mac80211 tells the driver what to do
2. driver internal use (if applicable)
3. TX status information - driver tells mac80211 what happened

void **ieee80211_tx_info_clear_status**(struct *ieee80211_tx_info* * *info*)
clear TX status

Parameters

struct ieee80211_tx_info * info The *struct ieee80211_tx_info* to be cleared.

Description

When the driver passes an skb back to mac80211, it must report a number of things in TX status. This function clears everything in the TX status but the rate control information (it does clear the count since you need to fill that in anyway).

NOTE

You can only use this function if you do NOT use info->driver_data! Use info->rate_driver_data instead if you need only the less space that allows.

void **ieee80211_rx**(struct *ieee80211_hw* * *hw*, struct sk_buff * *skb*)
receive frame

Parameters

struct ieee80211_hw * hw the hardware this frame came in on

struct sk_buff * skb the buffer to receive, owned by mac80211 after this call

Description

Use this function to hand received frames to mac80211. The receive buffer in **skb** must start with an IEEE 802.11 header. In case of a paged **skb** is used, the driver is recommended to put the ieee80211 header of the frame on the linear part of the **skb** to avoid memory allocation and/or memcpy by the stack.

This function may not be called in IRQ context. Calls to this function for a single hardware must be synchronized against each other. Calls to this function, *ieee80211_rx_ni()* and *ieee80211_rx_irqsafe()* may not be mixed for a single hardware. Must not run concurrently with *ieee80211_tx_status()* or *ieee80211_tx_status_ni()*.

In process context use instead *ieee80211_rx_ni()*.

void **ieee80211_rx_ni**(struct *ieee80211_hw* * *hw*, struct sk_buff * *skb*)
receive frame (in process context)

Parameters

struct ieee80211_hw * hw the hardware this frame came in on

struct sk_buff * skb the buffer to receive, owned by mac80211 after this call

Description

Like *ieee80211_rx()* but can be called in process context (internally disables bottom halves).

Calls to this function, *ieee80211_rx()* and *ieee80211_rx_irqsafe()* may not be mixed for a single hardware. Must not run concurrently with *ieee80211_tx_status()* or *ieee80211_tx_status_ni()*.

void **ieee80211_rx_irqsafe**(struct *ieee80211_hw* * *hw*, struct sk_buff * *skb*)
receive frame

Parameters

struct ieee80211_hw * hw the hardware this frame came in on

struct sk_buff * skb the buffer to receive, owned by mac80211 after this call

Description

Like *ieee80211_rx()* but can be called in IRQ context (internally defers to a tasklet.)

Calls to this function, *ieee80211_rx()* or *ieee80211_rx_ni()* may not be mixed for a single hardware.Must not run concurrently with *ieee80211_tx_status()* or *ieee80211_tx_status_ni()*.

struct **ieee80211_tx_status**
extended tx staus info for rate control

Definition

```
struct ieee80211_tx_status {
  struct ieee80211_sta *sta;
  struct ieee80211_tx_info *info;
  struct sk_buff *skb;
};
```

Members

sta Station that the packet was transmitted for

info Basic tx status information

skb Packet skb (can be NULL if not provided by the driver)

void **ieee80211_tx_status**(struct *ieee80211_hw* * *hw*, struct sk_buff * *skb*)
transmit status callback

Parameters

struct ieee80211_hw * hw the hardware the frame was transmitted by

struct sk_buff * skb the frame that was transmitted, owned by mac80211 after this call

Description

Call this function for all transmitted frames after they have been transmitted. It is permissible to not call this function for multicast frames but this can affect statistics.

This function may not be called in IRQ context. Calls to this function for a single hardware must be synchronized against each other. Calls to this function, *ieee80211_tx_status_ni()* and *ieee80211_tx_status_irqsafe()* may not be mixed for a single hardware. Must not run concurrently with *ieee80211_rx()* or *ieee80211_rx_ni()*.

void **ieee80211_tx_status_ni**(struct *ieee80211_hw* * *hw*, struct sk_buff * *skb*)
transmit status callback (in process context)

Parameters

struct ieee80211_hw * hw the hardware the frame was transmitted by

struct sk_buff * skb the frame that was transmitted, owned by mac80211 after this call

Description

Like *ieee80211_tx_status()* but can be called in process context.

Calls to this function, *ieee80211_tx_status()* and *ieee80211_tx_status_irqsafe()* may not be mixed for a single hardware.

void **ieee80211_tx_status_irqsafe**(struct *ieee80211_hw* * *hw*, struct sk_buff * *skb*)
IRQ-safe transmit status callback

Parameters

struct ieee80211_hw * hw the hardware the frame was transmitted by

struct sk_buff * skb the frame that was transmitted, owned by mac80211 after this call

Description

Like *ieee80211_tx_status()* but can be called in IRQ context (internally defers to a tasklet.)

Calls to this function, *ieee80211_tx_status()* and *ieee80211_tx_status_ni()* may not be mixed for a single hardware.

void **ieee80211_rts_get**(struct *ieee80211_hw* * *hw*, struct *ieee80211_vif* * *vif*, const void * *frame*, size_t *frame_len*, const struct *ieee80211_tx_info* * *frame_txctl*, struct ieee80211_rts * *rts*)
RTS frame generation function

Parameters

struct ieee80211_hw * hw pointer obtained from *ieee80211_alloc_hw()*.

struct ieee80211_vif * vif *struct ieee80211_vif* pointer from the add_interface callback.

const void * frame pointer to the frame that is going to be protected by the RTS.

size_t frame_len the frame length (in octets).

const struct ieee80211_tx_info * frame_txctl *struct ieee80211_tx_info* of the frame.

struct ieee80211_rts * rts The buffer where to store the RTS frame.

Description

If the RTS frames are generated by the host system (i.e., not in hardware/firmware), the low-level driver uses this function to receive the next RTS frame from the 802.11 code. The low-level is responsible for calling this function before and RTS frame is needed.

__le16 **ieee80211_rts_duration**(struct *ieee80211_hw* * *hw*, struct *ieee80211_vif* * *vif*, size_t *frame_len*, const struct *ieee80211_tx_info* * *frame_txctl*)
Get the duration field for an RTS frame

Parameters

struct ieee80211_hw * hw pointer obtained from *ieee80211_alloc_hw()*.

struct ieee80211_vif * vif *struct ieee80211_vif* pointer from the add_interface callback.

size_t frame_len the length of the frame that is going to be protected by the RTS.

const struct ieee80211_tx_info * frame_txctl *struct ieee80211_tx_info* of the frame.

Description

If the RTS is generated in firmware, but the host system must provide the duration field, the low-level driver uses this function to receive the duration field value in little-endian byteorder.

Return

The duration.

void **ieee80211_ctstoself_get**(struct *ieee80211_hw* * *hw*, struct *ieee80211_vif* * *vif*, const void * *frame*, size_t *frame_len*, const struct *ieee80211_tx_info* * *frame_txctl*, struct ieee80211_cts * *cts*)
CTS-to-self frame generation function

Parameters

struct ieee80211_hw * hw pointer obtained from *ieee80211_alloc_hw()*.

struct ieee80211_vif * vif *struct ieee80211_vif* pointer from the add_interface callback.

const void * frame pointer to the frame that is going to be protected by the CTS-to-self.

size_t frame_len the frame length (in octets).

const struct ieee80211_tx_info * frame_txctl *struct ieee80211_tx_info* of the frame.

struct ieee80211_cts * cts The buffer where to store the CTS-to-self frame.

Description

If the CTS-to-self frames are generated by the host system (i.e., not in hardware/firmware), the low-level driver uses this function to receive the next CTS-to-self frame from the 802.11 code. The low-level is responsible for calling this function before and CTS-to-self frame is needed.

__le16 **ieee80211_ctstoself_duration**(struct *ieee80211_hw* * *hw*, struct *ieee80211_vif* * *vif*, size_t *frame_len*, const struct *ieee80211_tx_info* * *frame_txctl*)

Get the duration field for a CTS-to-self frame

Parameters

struct ieee80211_hw * hw pointer obtained from *ieee80211_alloc_hw()*.

struct ieee80211_vif * vif *struct ieee80211_vif* pointer from the add_interface callback.

size_t frame_len the length of the frame that is going to be protected by the CTS-to-self.

const struct ieee80211_tx_info * frame_txctl *struct ieee80211_tx_info* of the frame.

Description

If the CTS-to-self is generated in firmware, but the host system must provide the duration field, the low-level driver uses this function to receive the duration field value in little-endian byteorder.

Return

The duration.

__le16 **ieee80211_generic_frame_duration**(struct *ieee80211_hw* * *hw*, struct *ieee80211_vif* * *vif*, enum nl80211_band *band*, size_t *frame_len*, struct *ieee80211_rate* * *rate*)

Calculate the duration field for a frame

Parameters

struct ieee80211_hw * hw pointer obtained from *ieee80211_alloc_hw()*.

struct ieee80211_vif * vif *struct ieee80211_vif* pointer from the add_interface callback.

enum nl80211_band band the band to calculate the frame duration on

size_t frame_len the length of the frame.

struct ieee80211_rate * rate the rate at which the frame is going to be transmitted.

Description

Calculate the duration field of some generic frame, given its length and transmission rate (in 100kbps).

Return

The duration.

void **ieee80211_wake_queue**(struct *ieee80211_hw* * *hw*, int *queue*)

wake specific queue

Parameters

struct ieee80211_hw * hw pointer as obtained from *ieee80211_alloc_hw()*.

int queue queue number (counted from zero).

Description

Drivers should use this function instead of netif_wake_queue.

void **ieee80211_stop_queue**(struct *ieee80211_hw* * *hw*, int *queue*)

stop specific queue

Parameters

struct ieee80211_hw * hw pointer as obtained from *ieee80211_alloc_hw()*.

int queue queue number (counted from zero).

Description

Drivers should use this function instead of netif_stop_queue.

void **ieee80211_wake_queues**(struct *ieee80211_hw* * *hw*)
wake all queues

Parameters

struct ieee80211_hw * hw pointer as obtained from *ieee80211_alloc_hw()*.

Description

Drivers should use this function instead of netif_wake_queue.

void **ieee80211_stop_queues**(struct *ieee80211_hw* * *hw*)
stop all queues

Parameters

struct ieee80211_hw * hw pointer as obtained from *ieee80211_alloc_hw()*.

Description

Drivers should use this function instead of netif_stop_queue.

int **ieee80211_queue_stopped**(struct *ieee80211_hw* * *hw*, int *queue*)
test status of the queue

Parameters

struct ieee80211_hw * hw pointer as obtained from *ieee80211_alloc_hw()*.

int queue queue number (counted from zero).

Description

Drivers should use this function instead of netif_stop_queue.

Return

`true` if the queue is stopped. `false` otherwise.

Frame filtering

mac80211 requires to see many management frames for proper operation, and users may want to see many more frames when in monitor mode. However, for best CPU usage and power consumption, having as few frames as possible percolate through the stack is desirable. Hence, the hardware should filter as much as possible.

To achieve this, mac80211 uses filter flags (see below) to tell the driver's `configure_filter()` function which frames should be passed to mac80211 and which should be filtered out.

Before `configure_filter()` is invoked, the `prepare_multicast()` callback is invoked with the parameters **mc_count** and **mc_list** for the combined multicast address list of all virtual interfaces. It's use is optional, and it returns a u64 that is passed to `configure_filter()`. Additionally, `configure_filter()` has the arguments **changed_flags** telling which flags were changed and **total_flags** with the new flag states.

If your device has no multicast address filters your driver will need to check both the `FIF_ALLMULTI` flag and the **mc_count** parameter to see whether multicast frames should be accepted or dropped.

All unsupported flags in **total_flags** must be cleared. Hardware does not support a flag if it is incapable of _passing_ the frame to the stack. Otherwise the driver must ignore the flag, but not clear it. You must _only_ clear the flag (announce no support for the flag to mac80211) if you are not able to pass the packet type to the stack (so the hardware always filters it). So for example, you should clear **FIF_CONTROL**, if your hardware always filters control frames. If your hardware always passes control frames to the kernel and is incapable of filtering them, you do _not_ clear the **FIF_CONTROL** flag. This rule applies to all other FIF flags as well.

enum **`ieee80211_filter_flags`**
hardware filter flags

Constants

`FIF_ALLMULTI` pass all multicast frames, this is used if requested by the user or if the hardware is not capable of filtering by multicast address.

`FIF_FCSFAIL` pass frames with failed FCS (but you need to set the RX_FLAG_FAILED_FCS_CRC for them)

`FIF_PLCPFAIL` pass frames with failed PLCP CRC (but you need to set the RX_FLAG_FAILED_PLCP_CRC for them

`FIF_BCN_PRBRESP_PROMISC` This flag is set during scanning to indicate to the hardware that it should not filter beacons or probe responses by BSSID. Filtering them can greatly reduce the amount of processing mac80211 needs to do and the amount of CPU wakeups, so you should honour this flag if possible.

`FIF_CONTROL` pass control frames (except for PS Poll) addressed to this station

`FIF_OTHER_BSS` pass frames destined to other BSSes

`FIF_PSPOLL` pass PS Poll frames

`FIF_PROBE_REQ` pass probe request frames

Description

These flags determine what the filter in hardware should be programmed to let through and what should not be passed to the stack. It is always safe to pass more frames than requested, but this has negative impact on power consumption.

The mac80211 workqueue

mac80211 provides its own workqueue for drivers and internal mac80211 use. The workqueue is a single threaded workqueue and can only be accessed by helpers for sanity checking. Drivers must ensure all work added onto the mac80211 workqueue should be cancelled on the driver `stop()` callback.

mac80211 will flushed the workqueue upon interface removal and during suspend.

All work performed on the mac80211 workqueue must not acquire the RTNL lock.

void **`ieee80211_queue_work`**(struct *ieee80211_hw* * *hw*, struct work_struct * *work*)
add work onto the mac80211 workqueue

Parameters

`struct ieee80211_hw * hw` the hardware struct for the interface we are adding work for

`struct work_struct * work` the work we want to add onto the mac80211 workqueue

Description

Drivers and mac80211 use this to add work onto the mac80211 workqueue. This helper ensures drivers are not queueing work when they should not be.

void **`ieee80211_queue_delayed_work`**(struct *ieee80211_hw* * *hw*, struct delayed_work * *dwork*, unsigned long *delay*)
add work onto the mac80211 workqueue

Parameters

`struct ieee80211_hw * hw` the hardware struct for the interface we are adding work for

`struct delayed_work * dwork` delayable work to queue onto the mac80211 workqueue

`unsigned long delay` number of jiffies to wait before queueing

Description

Drivers and mac80211 use this to queue delayed work onto the mac80211 workqueue.

mac80211 subsystem (advanced)

Information contained within this part of the book is of interest only for advanced interaction of mac80211 with drivers to exploit more hardware capabilities and improve performance.

LED support

Mac80211 supports various ways of blinking LEDs. Wherever possible, device LEDs should be exposed as LED class devices and hooked up to the appropriate trigger, which will then be triggered appropriately by mac80211.

const char * **ieee80211_get_tx_led_name**(struct *ieee80211_hw* * *hw*)
get name of TX LED

Parameters

struct ieee80211_hw * hw the hardware to get the LED trigger name for

Description

mac80211 creates a transmit LED trigger for each wireless hardware that can be used to drive LEDs if your driver registers a LED device. This function returns the name (or NULL if not configured for LEDs) of the trigger so you can automatically link the LED device.

Return

The name of the LED trigger. NULL if not configured for LEDs.

const char * **ieee80211_get_rx_led_name**(struct *ieee80211_hw* * *hw*)
get name of RX LED

Parameters

struct ieee80211_hw * hw the hardware to get the LED trigger name for

Description

mac80211 creates a receive LED trigger for each wireless hardware that can be used to drive LEDs if your driver registers a LED device. This function returns the name (or NULL if not configured for LEDs) of the trigger so you can automatically link the LED device.

Return

The name of the LED trigger. NULL if not configured for LEDs.

const char * **ieee80211_get_assoc_led_name**(struct *ieee80211_hw* * *hw*)
get name of association LED

Parameters

struct ieee80211_hw * hw the hardware to get the LED trigger name for

Description

mac80211 creates a association LED trigger for each wireless hardware that can be used to drive LEDs if your driver registers a LED device. This function returns the name (or NULL if not configured for LEDs) of the trigger so you can automatically link the LED device.

Return

The name of the LED trigger. NULL if not configured for LEDs.

const char * **ieee80211_get_radio_led_name**(struct *ieee80211_hw* * *hw*)
get name of radio LED

Parameters

struct ieee80211_hw * hw the hardware to get the LED trigger name for

Description

mac80211 creates a radio change LED trigger for each wireless hardware that can be used to drive LEDs if your driver registers a LED device. This function returns the name (or NULL if not configured for LEDs) of the trigger so you can automatically link the LED device.

Return

The name of the LED trigger. NULL if not configured for LEDs.

struct **ieee80211_tpt_blink**
throughput blink description

Definition

```
struct ieee80211_tpt_blink {
  int throughput;
  int blink_time;
};
```

Members

throughput throughput in Kbit/sec

blink_time blink time in milliseconds (full cycle, ie. one off + one on period)

enum **ieee80211_tpt_led_trigger_flags**
throughput trigger flags

Constants

IEEE80211_TPT_LEDTRIG_FL_RADIO enable blinking with radio

IEEE80211_TPT_LEDTRIG_FL_WORK enable blinking when working

IEEE80211_TPT_LEDTRIG_FL_CONNECTED enable blinking when at least one interface is connected in some way, including being an AP

const char * **ieee80211_create_tpt_led_trigger**(struct *ieee80211_hw* * *hw*, unsigned int *flags*, const struct *ieee80211_tpt_blink* * *blink_table*, unsigned int *blink_table_len*)
create throughput LED trigger

Parameters

struct ieee80211_hw * hw the hardware to create the trigger for

unsigned int flags trigger flags, see *enum ieee80211_tpt_led_trigger_flags*

const struct ieee80211_tpt_blink * blink_table the blink table – needs to be ordered by throughput

unsigned int blink_table_len size of the blink table

Return

NULL (in case of error, or if no LED triggers are configured) or the name of the new trigger.

Note

This function must be called before *ieee80211_register_hw()*.

Hardware crypto acceleration

mac80211 is capable of taking advantage of many hardware acceleration designs for encryption and decryption operations.

The `set_key()` callback in the *struct ieee80211_ops* for a given device is called to enable hardware acceleration of encryption and decryption. The callback takes a **sta** parameter that will be NULL for default keys or keys used for transmission only, or point to the station information for the peer for individual keys. Multiple transmission keys with the same key index may be used when VLANs are configured for an access point.

When transmitting, the TX control data will use the **hw_key_idx** selected by the driver by modifying the *struct ieee80211_key_conf* pointed to by the **key** parameter to the `set_key()` function.

The `set_key()` call for the SET_KEY command should return 0 if the key is now in use, `-EOPNOTSUPP` or `-ENOSPC` if it couldn't be added; if you return 0 then hw_key_idx must be assigned to the hardware key index, you are free to use the full u8 range.

Note that in the case that the **IEEE80211_HW_SW_CRYPTO_CONTROL** flag is set, mac80211 will not automatically fall back to software crypto if enabling hardware crypto failed. The `set_key()` call may also return the value 1 to permit this specific key/algorithm to be done in software.

When the cmd is `DISABLE_KEY` then it must succeed.

Note that it is permissible to not decrypt a frame even if a key for it has been uploaded to hardware, the stack will not make any decision based on whether a key has been uploaded or not but rather based on the receive flags.

The *struct ieee80211_key_conf* structure pointed to by the **key** parameter is guaranteed to be valid until another call to `set_key()` removes it, but it can only be used as a cookie to differentiate keys.

In TKIP some HW need to be provided a phase 1 key, for RX decryption acceleration (i.e. iwlwifi). Those drivers should provide update_tkip_key handler. The `update_tkip_key()` call updates the driver with the new phase 1 key. This happens every time the iv16 wraps around (every 65536 packets). The `set_key()` call will happen only once for each key (unless the AP did rekeying), it will not include a valid phase 1 key. The valid phase 1 key is provided by update_tkip_key only. The trigger that makes mac80211 call this handler is software decryption with wrap around of iv16.

The `set_default_unicast_key()` call updates the default WEP key index configured to the hardware for WEP encryption type. This is required for devices that support offload of data packets (e.g. ARP responses).

enum **`set_key_cmd`**
key command

Constants

`SET_KEY` a key is set

`DISABLE_KEY` a key must be disabled

Description

Used with the `set_key()` callback in *struct ieee80211_ops*, this indicates whether a key is being removed or added.

struct **`ieee80211_key_conf`**
key information

Definition

```
struct ieee80211_key_conf {
  atomic64_t tx_pn;
  u32 cipher;
  u8 icv_len;
  u8 iv_len;
  u8 hw_key_idx;
  s8 keyidx;
  u16 flags;
  u8 keylen;
  u8 key[0];
};
```

Members

`tx_pn` PN used for TX keys, may be used by the driver as well if it needs to do software PN assignment by itself (e.g. due to TSO)

`cipher` The key's cipher suite selector.

`icv_len` The ICV length for this key type

`iv_len` The IV length for this key type

`hw_key_idx` To be set by the driver, this is the key index the driver wants to be given when a frame is transmitted and needs to be encrypted in hardware.

`keyidx` the key index (0-3)

`flags` key flags, see *`enum ieee80211_key_flags`*.

`keylen` key material length

`key` key material. For ALG_TKIP the key is encoded as a 256-bit (32 byte) data block: - Temporal Encryption Key (128 bits) - Temporal Authenticator Tx MIC Key (64 bits) - Temporal Authenticator Rx MIC Key (64 bits)

Description

This key information is given by mac80211 to the driver by the `set_key()` callback in *`struct ieee80211_ops`*.

enum **`ieee80211_key_flags`**
key flags

Constants

`IEEE80211_KEY_FLAG_GENERATE_IV_MGMT` This flag should be set by the driver for a CCMP/GCMP key to indicate that is requires IV generation only for managment frames (MFP).

`IEEE80211_KEY_FLAG_GENERATE_IV` This flag should be set by the driver to indicate that it requires IV generation for this particular key. Setting this flag does not necessarily mean that SKBs will have sufficient tailroom for ICV or MIC.

`IEEE80211_KEY_FLAG_GENERATE_MMIC` This flag should be set by the driver for a TKIP key if it requires Michael MIC generation in software.

`IEEE80211_KEY_FLAG_PAIRWISE` Set by mac80211, this flag indicates that the key is pairwise rather then a shared key.

`IEEE80211_KEY_FLAG_SW_MGMT_TX` This flag should be set by the driver for a CCMP/GCMP key if it requires CCMP/GCMP encryption of management frames (MFP) to be done in software.

`IEEE80211_KEY_FLAG_PUT_IV_SPACE` This flag should be set by the driver if space should be prepared for the IV, but the IV itself should not be generated. Do not set together with **IEEE80211_KEY_FLAG_GENERATE_IV** on the same key. Setting this flag does not necessarily mean that SKBs will have sufficient tailroom for ICV or MIC.

`IEEE80211_KEY_FLAG_RX_MGMT` This key will be used to decrypt received management frames. The flag can help drivers that have a hardware crypto implementation that doesn't deal with management frames properly by allowing them to not upload the keys to hardware and fall back to software crypto. Note that this flag deals only with RX, if your crypto engine can't deal with TX you can also set the `IEEE80211_KEY_FLAG_SW_MGMT_TX` flag to encrypt such frames in SW.

`IEEE80211_KEY_FLAG_RESERVE_TAILROOM` This flag should be set by the driver for a key to indicate that sufficient tailroom must always be reserved for ICV or MIC, even when HW encryption is enabled.

IEEE80211_KEY_FLAG_PUT_MIC_SPACE This flag should be set by the driver for a TKIP key if it only requires MIC space. Do not set together with **IEEE80211_KEY_FLAG_GENERATE_MMIC** on the same key.

Description

These flags are used for communication about keys between the driver and mac80211, with the **flags** parameter of *struct ieee80211_key_conf*.

void **ieee80211_get_tkip_p1k**(struct *ieee80211_key_conf* * *keyconf*, struct sk_buff * *skb*, u16 * *p1k*)
get a TKIP phase 1 key

Parameters

struct ieee80211_key_conf * keyconf the parameter passed with the set key

struct sk_buff * skb the packet to take the IV32 value from that will be encrypted with this P1K

u16 * p1k a buffer to which the key will be written, as 5 u16 values

Description

This function returns the TKIP phase 1 key for the IV32 taken from the given packet.

void **ieee80211_get_tkip_p1k_iv**(struct *ieee80211_key_conf* * *keyconf*, u32 *iv32*, u16 * *p1k*)
get a TKIP phase 1 key for IV32

Parameters

struct ieee80211_key_conf * keyconf the parameter passed with the set key

u32 iv32 IV32 to get the P1K for

u16 * p1k a buffer to which the key will be written, as 5 u16 values

Description

This function returns the TKIP phase 1 key for the given IV32.

void **ieee80211_get_tkip_p2k**(struct *ieee80211_key_conf* * *keyconf*, struct sk_buff * *skb*, u8 * *p2k*)
get a TKIP phase 2 key

Parameters

struct ieee80211_key_conf * keyconf the parameter passed with the set key

struct sk_buff * skb the packet to take the IV32/IV16 values from that will be encrypted with this key

u8 * p2k a buffer to which the key will be written, 16 bytes

Description

This function computes the TKIP RC4 key for the IV values in the packet.

Powersave support

mac80211 has support for various powersave implementations.

First, it can support hardware that handles all powersaving by itself, such hardware should simply set the `IEEE80211_HW_SUPPORTS_PS` hardware flag. In that case, it will be told about the desired powersave mode with the `IEEE80211_CONF_PS` flag depending on the association status. The hardware must take care of sending nullfunc frames when necessary, i.e. when entering and leaving powersave mode. The hardware is required to look at the AID in beacons and signal to the AP that it woke up when it finds traffic directed to it.

`IEEE80211_CONF_PS` flag enabled means that the powersave mode defined in IEEE 802.11-2007 section 11.2 is enabled. This is not to be confused with hardware wakeup and sleep states. Driver is responsible for waking up the hardware before issuing commands to the hardware and putting it back to sleep at appropriate times.

When PS is enabled, hardware needs to wakeup for beacons and receive the buffered multicast/broadcast frames after the beacon. Also it must be possible to send frames and receive the acknowledment frame.

Other hardware designs cannot send nullfunc frames by themselves and also need software support for parsing the TIM bitmap. This is also supported by mac80211 by combining the `IEEE80211_HW_SUPPORTS_PS` and `IEEE80211_HW_PS_NULLFUNC_STACK` flags. The hardware is of course still required to pass up beacons. The hardware is still required to handle waking up for multicast traffic; if it cannot the driver must handle that as best as it can, mac80211 is too slow to do that.

Dynamic powersave is an extension to normal powersave in which the hardware stays awake for a user-specified period of time after sending a frame so that reply frames need not be buffered and therefore delayed to the next wakeup. It's compromise of getting good enough latency when there's data traffic and still saving significantly power in idle periods.

Dynamic powersave is simply supported by mac80211 enabling and disabling PS based on traffic. Driver needs to only set `IEEE80211_HW_SUPPORTS_PS` flag and mac80211 will handle everything automatically. Additionally, hardware having support for the dynamic PS feature may set the `IEEE80211_HW_SUPPORTS_DYNAMIC_PS` flag to indicate that it can support dynamic PS mode itself. The driver needs to look at the **dynamic_ps_timeout** hardware configuration value and use it that value whenever `IEEE80211_CONF_PS` is set. In this case mac80211 will disable dynamic PS feature in stack and will just keep `IEEE80211_CONF_PS` enabled whenever user has enabled powersave.

Driver informs U-APSD client support by enabling `IEEE80211_VIF_SUPPORTS_UAPSD` flag. The mode is configured through the uapsd parameter in `conf_tx()` operation. Hardware needs to send the QoS Nullfunc frames and stay awake until the service period has ended. To utilize U-APSD, dynamic powersave is disabled for voip AC and all frames from that AC are transmitted with powersave enabled.

Note: U-APSD client mode is not yet supported with `IEEE80211_HW_PS_NULLFUNC_STACK`.

Beacon filter support

Some hardware have beacon filter support to reduce host cpu wakeups which will reduce system power consumption. It usually works so that the firmware creates a checksum of the beacon but omits all constantly changing elements (TSF, TIM etc). Whenever the checksum changes the beacon is forwarded to the host, otherwise it will be just dropped. That way the host will only receive beacons where some relevant information (for example ERP protection or WMM settings) have changed.

Beacon filter support is advertised with the `IEEE80211_VIF_BEACON_FILTER` interface capability. The driver needs to enable beacon filter support whenever power save is enabled, that is `IEEE80211_CONF_PS` is set. When power save is enabled, the stack will not check for beacon loss and the driver needs to notify about loss of beacons with *`ieee80211_beacon_loss()`*.

The time (or number of beacons missed) until the firmware notifies the driver of a beacon loss event (which in turn causes the driver to call *`ieee80211_beacon_loss()`*) should be configurable and will be controlled by mac80211 and the roaming algorithm in the future.

Since there may be constantly changing information elements that nothing in the software stack cares about, we will, in the future, have mac80211 tell the driver which information elements are interesting in the sense that we want to see changes in them. This will include

- a list of information element IDs
- a list of OUIs for the vendor information element

Ideally, the hardware would filter out any beacons without changes in the requested elements, but if it cannot support that it may, at the expense of some efficiency, filter out only a subset. For example, if the device doesn't support checking for OUIs it should pass up all changes in all vendor information elements.

Note that change, for the sake of simplification, also includes information elements appearing or disappearing from the beacon.

Some hardware supports an "ignore list" instead, just make sure nothing that was requested is on the ignore list, and include commonly changing information element IDs in the ignore list, for example 11 (BSS load) and the various vendor-assigned IEs with unknown contents (128, 129, 133-136, 149, 150, 155, 156, 173, 176, 178, 179, 219); for forward compatibility it could also include some currently unused IDs.

In addition to these capabilities, hardware should support notifying the host of changes in the beacon RSSI. This is relevant to implement roaming when no traffic is flowing (when traffic is flowing we see the RSSI of the received data packets). This can consist in notifying the host when the RSSI changes significantly or when it drops below or rises above configurable thresholds. In the future these thresholds will also be configured by mac80211 (which gets them from userspace) to implement them as the roaming algorithm requires.

If the hardware cannot implement this, the driver should ask it to periodically pass beacon frames to the host so that software can do the signal strength threshold checking.

void **ieee80211_beacon_loss**(struct *ieee80211_vif* * *vif*)
inform hardware does not receive beacons

Parameters

struct ieee80211_vif * vif *struct ieee80211_vif* pointer from the add_interface callback.

Description

When beacon filtering is enabled with IEEE80211_VIF_BEACON_FILTER and IEEE80211_CONF_PS is set, the driver needs to inform whenever the hardware is not receiving beacons with this function.

Multiple queues and QoS support

TBD

struct **ieee80211_tx_queue_params**
transmit queue configuration

Definition

```
struct ieee80211_tx_queue_params {
  u16 txop;
  u16 cw_min;
  u16 cw_max;
  u8 aifs;
  bool acm;
  bool uapsd;
};
```

Members

txop maximum burst time in units of 32 usecs, 0 meaning disabled

cw_min minimum contention window [a value of the form 2^n-1 in the range 1..32767]

cw_max maximum contention window [like **cw_min**]

aifs arbitration interframe space [0..255]

acm is mandatory admission control required for the access category

uapsd is U-APSD mode enabled for the queue

Description

The information provided in this structure is required for QoS transmit queue configuration. Cf. IEEE 802.11 7.3.2.29.

Access point mode support

TBD

Some parts of the if_conf should be discussed here instead

Insert notes about VLAN interfaces with hw crypto here or in the hw crypto chapter.

support for powersaving clients

In order to implement AP and P2P GO modes, mac80211 has support for client powersaving, both "legacy" PS (PS-Poll/null data) and uAPSD. There currently is no support for sAPSD.

There is one assumption that mac80211 makes, namely that a client will not poll with PS-Poll and trigger with uAPSD at the same time. Both are supported, and both can be used by the same client, but they can't be used concurrently by the same client. This simplifies the driver code.

The first thing to keep in mind is that there is a flag for complete driver implementation: `IEEE80211_HW_AP_LINK_PS`. If this flag is set, mac80211 expects the driver to handle most of the state machine for powersaving clients and will ignore the PM bit in incoming frames. Drivers then use *`ieee80211_sta_ps_transition()`* to inform mac80211 of stations' powersave transitions. In this mode, mac80211 also doesn't handle PS-Poll/uAPSD.

In the mode without `IEEE80211_HW_AP_LINK_PS`, mac80211 will check the PM bit in incoming frames for client powersave transitions. When a station goes to sleep, we will stop transmitting to it. There is, however, a race condition: a station might go to sleep while there is data buffered on hardware queues. If the device has support for this it will reject frames, and the driver should give the frames back to mac80211 with the `IEEE80211_TX_STAT_TX_FILTERED` flag set which will cause mac80211 to retry the frame when the station wakes up. The driver is also notified of powersave transitions by calling its **sta_notify** callback.

When the station is asleep, it has three choices: it can wake up, it can PS-Poll, or it can possibly start a uAPSD service period. Waking up is implemented by simply transmitting all buffered (and filtered) frames to the station. This is the easiest case. When the station sends a PS-Poll or a uAPSD trigger frame, mac80211 will inform the driver of this with the **allow_buffered_frames** callback; this callback is optional. mac80211 will then transmit the frames as usual and set the `IEEE80211_TX_CTL_NO_PS_BUFFER` on each frame. The last frame in the service period (or the only response to a PS-Poll) also has `IEEE80211_TX_STATUS_EOSP` set to indicate that it ends the service period; as this frame must have TX status report it also sets `IEEE80211_TX_CTL_REQ_TX_STATUS`. When TX status is reported for this frame, the service period is marked has having ended and a new one can be started by the peer.

Additionally, non-bufferable MMPDUs can also be transmitted by mac80211 with the `IEEE80211_TX_CTL_NO_PS_BUFFER` set in them.

Another race condition can happen on some devices like iwlwifi when there are frames queued for the station and it wakes up or polls; the frames that are already queued could end up being transmitted first instead, causing reordering and/or wrong processing of the EOSP. The cause is that allowing frames to be transmitted to a certain station is out-of-band communication to the device. To allow this problem to be solved, the driver can call *`ieee80211_sta_block_awake()`* if frames are buffered when it is notified that the station went to sleep. When all these frames have been filtered (see above), it must call the function again to indicate that the station is no longer blocked.

If the driver buffers frames in the driver for aggregation in any way, it must use the *`ieee80211_sta_set_buffered()`* call when it is notified of the station going to sleep to inform mac80211 of any TIDs that have frames buffered. Note that when a station wakes up this information is reset (hence the requirement to call it when informed of the station going to sleep). Then, when a service period starts for any reason, **release_buffered_frames** is called with the number of frames to be released and which TIDs they are to come from. In this case, the driver is responsible for setting the EOSP (for uAPSD) and MORE_DATA bits in the released frames, to help the **more_data** parameter is passed to tell the driver if there is more data on other TIDs – the TIDs to release frames from are ignored since mac80211 doesn't know how many frames the buffers for those TIDs contain.

If the driver also implement GO mode, where absence periods may shorten service periods (or abort PS-Poll responses), it must filter those response frames except in the case of frames that are buffered in the driver – those must remain buffered to avoid reordering. Because it is possible that no frames are released in this case, the driver must call *`ieee80211_sta_eosp()`* to indicate to mac80211 that the service period ended anyway.

Finally, if frames from multiple TIDs are released from mac80211 but the driver might reorder them, it must clear & set the flags appropriately (only the last frame may have `IEEE80211_TX_STATUS_EOSP`) and also take care of the EOSP and MORE_DATA bits in the frame. The driver may also use *`ieee80211_sta_eosp()`* in this case.

Note that if the driver ever buffers frames other than QoS-data frames, it must take care to never send a non-QoS-data frame as the last frame in a service period, adding a QoS-nulldata frame after a non-QoS-data frame if needed.

struct sk_buff * **`ieee80211_get_buffered_bc`**(struct *ieee80211_hw* * *hw*, struct *ieee80211_vif* * *vif*)
accessing buffered broadcast and multicast frames

Parameters

`struct ieee80211_hw * hw` pointer as obtained from *`ieee80211_alloc_hw()`*.

`struct ieee80211_vif * vif` *`struct ieee80211_vif`* pointer from the add_interface callback.

Description

Function for accessing buffered broadcast and multicast frames. If hardware/firmware does not implement buffering of broadcast/multicast frames when power saving is used, 802.11 code buffers them in the host memory. The low-level driver uses this function to fetch next buffered frame. In most cases, this is used when generating beacon frame.

Return

A pointer to the next buffered skb or NULL if no more buffered frames are available.

Note

buffered frames are returned only after DTIM beacon frame was generated with *`ieee80211_beacon_get()`* and the low-level driver must thus call *`ieee80211_beacon_get()`* first. *`ieee80211_get_buffered_bc()`* returns NULL if the previous generated beacon was not DTIM, so the low-level driver does not need to check for DTIM beacons separately and should be able to use common code for all beacons.

struct sk_buff * **`ieee80211_beacon_get`**(struct *ieee80211_hw* * *hw*, struct *ieee80211_vif* * *vif*)
beacon generation function

Parameters

`struct ieee80211_hw * hw` pointer obtained from *`ieee80211_alloc_hw()`*.

`struct ieee80211_vif * vif` *`struct ieee80211_vif`* pointer from the add_interface callback.

Description

See `ieee80211_beacon_get_tim()`.

Return

See `ieee80211_beacon_get_tim()`.

void **`ieee80211_sta_eosp`**(struct *ieee80211_sta* * *pubsta*)
notify mac80211 about end of SP

Parameters

`struct ieee80211_sta * pubsta` the station

Description

When a device transmits frames in a way that it can't tell mac80211 in the TX status about the EOSP, it must clear the `IEEE80211_TX_STATUS_EOSP` bit and call this function instead. This applies for PS-Poll as well as uAPSD.

Note that just like with `_tx_status()` and `_rx()` drivers must not mix calls to irqsafe/non-irqsafe versions, this function must not be mixed with those either. Use the all irqsafe, or all non-irqsafe, don't mix!

NB: the _irqsafe version of this function doesn't exist, no driver needs it right now. Don't call this function if you'd need the _irqsafe version, look at the git history and restore the _irqsafe version!

enum **ieee80211_frame_release_type**
frame release reason

Constants

IEEE80211_FRAME_RELEASE_PSPOLL frame released for PS-Poll

IEEE80211_FRAME_RELEASE_UAPSD frame(s) released due to frame received on trigger-enabled AC

int **ieee80211_sta_ps_transition**(struct *ieee80211_sta* * *sta*, bool *start*)
PS transition for connected sta

Parameters

struct ieee80211_sta * sta currently connected sta

bool start start or stop PS

Description

When operating in AP mode with the IEEE80211_HW_AP_LINK_PS flag set, use this function to inform mac80211 about a connected station entering/leaving PS mode.

This function may not be called in IRQ context or with softirqs enabled.

Calls to this function for a single hardware must be synchronized against each other.

Return

0 on success. -EINVAL when the requested PS mode is already set.

int **ieee80211_sta_ps_transition_ni**(struct *ieee80211_sta* * *sta*, bool *start*)
PS transition for connected sta (in process context)

Parameters

struct ieee80211_sta * sta currently connected sta

bool start start or stop PS

Description

Like *ieee80211_sta_ps_transition()* but can be called in process context (internally disables bottom halves). Concurrent call restriction still applies.

Return

Like *ieee80211_sta_ps_transition()*.

void **ieee80211_sta_set_buffered**(struct *ieee80211_sta* * *sta*, u8 *tid*, bool *buffered*)
inform mac80211 about driver-buffered frames

Parameters

struct ieee80211_sta * sta *struct ieee80211_sta* pointer for the sleeping station

u8 tid the TID that has buffered frames

bool buffered indicates whether or not frames are buffered for this TID

Description

If a driver buffers frames for a powersave station instead of passing them back to mac80211 for retransmission, the station may still need to be told that there are buffered frames via the TIM bit.

This function informs mac80211 whether or not there are frames that are buffered in the driver for a given TID; mac80211 can then use this data to set the TIM bit (NOTE: This may call back into the driver's set_tim call! Beware of the locking!)

If all frames are released to the station (due to PS-poll or uAPSD) then the driver needs to inform mac80211 that there no longer are frames buffered. However, when the station wakes up mac80211 assumes that all buffered frames will be transmitted and clears this data, drivers need to make sure they inform mac80211 about all buffered frames on the sleep transition (`sta_notify()` with STA_NOTIFY_SLEEP).

Note that technically mac80211 only needs to know this per AC, not per TID, but since driver buffering will inevitably happen per TID (since it is related to aggregation) it is easier to make mac80211 map the TID to the AC as required instead of keeping track in all drivers that use this API.

void **ieee80211_sta_block_awake**(struct *ieee80211_hw* * *hw*, struct *ieee80211_sta* * *pubsta*, bool *block*)

block station from waking up

Parameters

struct ieee80211_hw * hw the hardware

struct ieee80211_sta * pubsta the station

bool block whether to block or unblock

Description

Some devices require that all frames that are on the queues for a specific station that went to sleep are flushed before a poll response or frames after the station woke up can be delivered to that it. Note that such frames must be rejected by the driver as filtered, with the appropriate status flag.

This function allows implementing this mode in a race-free manner.

To do this, a driver must keep track of the number of frames still enqueued for a specific station. If this number is not zero when the station goes to sleep, the driver must call this function to force mac80211 to consider the station to be asleep regardless of the station's actual state. Once the number of outstanding frames reaches zero, the driver must call this function again to unblock the station. That will cause mac80211 to be able to send ps-poll responses, and if the station queried in the meantime then frames will also be sent out as a result of this. Additionally, the driver will be notified that the station woke up some time after it is unblocked, regardless of whether the station actually woke up while blocked or not.

Supporting multiple virtual interfaces

TBD

Note: WDS with identical MAC address should almost always be OK

Insert notes about having multiple virtual interfaces with different MAC addresses here, note which configurations are supported by mac80211, add notes about supporting hw crypto with it.

void **ieee80211_iterate_active_interfaces**(struct *ieee80211_hw* * *hw*, u32 *iter_flags*, void (*iterator) (void **data*, u8 **mac*, struct *ieee80211_vif* **vif*, void * *data*)

iterate active interfaces

Parameters

struct ieee80211_hw * hw the hardware struct of which the interfaces should be iterated over

u32 iter_flags iteration flags, see enum `ieee80211_interface_iteration_flags`

void (*)(void *data, u8 *mac, struct ieee80211_vif *vif) iterator the iterator function to call

void * data first argument of the iterator function

Description

This function iterates over the interfaces associated with a given hardware that are currently active and calls the callback for them. This function allows the iterator function to sleep, when the iterator function is atomic **ieee80211_iterate_active_interfaces_atomic** can be used. Does not iterate over a new interface during `add_interface()`.

void **ieee80211_iterate_active_interfaces_atomic**(struct *ieee80211_hw * hw*, u32 *iter_flags*, void (*iterator) (void **data*, u8 **mac*, struct *ieee80211_vif *vif*, void * *data*)

iterate active interfaces

Parameters

struct ieee80211_hw * hw the hardware struct of which the interfaces should be iterated over

u32 iter_flags iteration flags, see enum `ieee80211_interface_iteration_flags`

void (*)(void *data, u8 *mac, struct ieee80211_vif *vif) iterator the iterator function to call, cannot sleep

void * data first argument of the iterator function

Description

This function iterates over the interfaces associated with a given hardware that are currently active and calls the callback for them. This function requires the iterator callback function to be atomic, if that is not desired, use **ieee80211_iterate_active_interfaces** instead. Does not iterate over a new interface during `add_interface()`.

Station handling

TODO

struct **ieee80211_sta**

station table entry

Definition

```
struct ieee80211_sta {
  u32 supp_rates[NUM_NL80211_BANDS];
  u8 addr[ETH_ALEN];
  u16 aid;
  struct ieee80211_sta_ht_cap ht_cap;
  struct ieee80211_sta_vht_cap vht_cap;
  u8 max_rx_aggregation_subframes;
  bool wme;
  u8 uapsd_queues;
  u8 max_sp;
  u8 rx_nss;
  enum ieee80211_sta_rx_bandwidth bandwidth;
  enum ieee80211_smps_mode smps_mode;
  struct ieee80211_sta_rates __rcu *rates;
  bool tdls;
  bool tdls_initiator;
  bool mfp;
  u8 max_amsdu_subframes;
  u16 max_amsdu_len;
  bool support_p2p_ps;
  u16 max_rc_amsdu_len;
  struct ieee80211_txq *txq[IEEE80211_NUM_TIDS];
  u8 drv_priv[0] ;
};
```

Members

supp_rates Bitmap of supported rates (per band)

addr MAC address

aid AID we assigned to the station if we're an AP

ht_cap HT capabilities of this STA; restricted to our own capabilities

vht_cap VHT capabilities of this STA; restricted to our own capabilities

max_rx_aggregation_subframes maximal amount of frames in a single AMPDU that this station is allowed to transmit to us. Can be modified by driver.

wme indicates whether the STA supports QoS/WME (if local devices does, otherwise always false)

uapsd_queues bitmap of queues configured for uapsd. Only valid if wme is supported. The bits order is like in IEEE80211_WMM_IE_STA_QOSINFO_AC_*.

max_sp max Service Period. Only valid if wme is supported.

rx_nss in HT/VHT, the maximum number of spatial streams the station can receive at the moment, changed by operating mode notifications and capabilities. The value is only valid after the station moves to associated state.

bandwidth current bandwidth the station can receive with

smps_mode current SMPS mode (off, static or dynamic)

rates rate control selection table

tdls indicates whether the STA is a TDLS peer

tdls_initiator indicates the STA is an initiator of the TDLS link. Only valid if the STA is a TDLS peer in the first place.

mfp indicates whether the STA uses management frame protection or not.

max_amsdu_subframes indicates the maximal number of MSDUs in a single A-MSDU. Taken from the Extended Capabilities element. 0 means unlimited.

max_amsdu_len indicates the maximal length of an A-MSDU in bytes. This field is always valid for packets with a VHT preamble. For packets with a HT preamble, additional limits apply:

- If the skb is transmitted as part of a BA agreement, the A-MSDU maximal size is min(max_amsdu_len, 4065) bytes.
- If the skb is not part of a BA aggreement, the A-MSDU maximal size is min(max_amsdu_len, 7935) bytes.

Both additional HT limits must be enforced by the low level driver. This is defined by the spec (IEEE 802.11-2012 section 8.3.2.2 NOTE 2).

support_p2p_ps indicates whether the STA supports P2P PS mechanism or not.

max_rc_amsdu_len Maximum A-MSDU size in bytes recommended by rate control.

txq per-TID data TX queues (if driver uses the TXQ abstraction)

drv_priv data area for driver use, will always be aligned to sizeof(void *), size is determined in hw information.

Description

A station table entry represents a station we are possibly communicating with. Since stations are RCU-managed in mac80211, any ieee80211_sta pointer you get access to must either be protected by `rcu_read_lock()` explicitly or implicitly, or you must take good care to not use such a pointer after a call to your sta_remove callback that removed it.

enum **sta_notify_cmd**
sta notify command

Constants

STA_NOTIFY_SLEEP a station is now sleeping

STA_NOTIFY_AWAKE a sleeping station woke up

Description

Used with the `sta_notify()` callback in *`struct ieee80211_ops`*, this indicates if an associated station made a power state transition.

struct *ieee80211_sta* * **`ieee80211_find_sta`**(struct *ieee80211_vif* * *vif*, const u8 * *addr*)
find a station

Parameters

`struct ieee80211_vif * vif` virtual interface to look for station on

`const u8 * addr` station's address

Return

The station, if found. `NULL` otherwise.

Note

This function must be called under RCU lock and the resulting pointer is only valid under RCU lock as well.

struct *ieee80211_sta* * **`ieee80211_find_sta_by_ifaddr`**(struct *ieee80211_hw* * *hw*, const u8 * *addr*, const u8 * *localaddr*)
find a station on hardware

Parameters

`struct ieee80211_hw * hw` pointer as obtained from *`ieee80211_alloc_hw()`*

`const u8 * addr` remote station's address

`const u8 * localaddr` local address (vif->sdata->vif.addr). Use NULL for 'any'.

Return

The station, if found. `NULL` otherwise.

Note

This function must be called under RCU lock and the resulting pointer is only valid under RCU lock as well.

NOTE

You may pass NULL for localaddr, but then you will just get the first STA that matches the remote address 'addr'. We can have multiple STA associated with multiple logical stations (e.g. consider a station connecting to another BSSID on the same AP hardware without disconnecting first). In this case, the result of this method with localaddr NULL is not reliable.

DO NOT USE THIS FUNCTION with localaddr NULL if at all possible.

Hardware scan offload

TBD

void **`ieee80211_scan_completed`**(struct *ieee80211_hw* * *hw*, struct cfg80211_scan_info * *info*)
completed hardware scan

Parameters

`struct ieee80211_hw * hw` the hardware that finished the scan

`struct cfg80211_scan_info * info` information about the completed scan

Description

When hardware scan offload is used (i.e. the `hw_scan()` callback is assigned) this function needs to be called by the driver to notify mac80211 that the scan finished. This function can be called from any context, including hardirq context.

Aggregation

TX A-MPDU aggregation

Aggregation on the TX side requires setting the hardware flag `IEEE80211_HW_AMPDU_AGGREGATION`. The driver will then be handed packets with a flag indicating A-MPDU aggregation. The driver or device is responsible for actually aggregating the frames, as well as deciding how many and which to aggregate.

When TX aggregation is started by some subsystem (usually the rate control algorithm would be appropriate) by calling the *ieee80211_start_tx_ba_session()* function, the driver will be notified via its **ampdu_action** function, with the IEEE80211_AMPDU_TX_START action.

In response to that, the driver is later required to call the *ieee80211_start_tx_ba_cb_irqsafe()* function, which will really start the aggregation session after the peer has also responded. If the peer responds negatively, the session will be stopped again right away. Note that it is possible for the aggregation session to be stopped before the driver has indicated that it is done setting it up, in which case it must not indicate the setup completion.

Also note that, since we also need to wait for a response from the peer, the driver is notified of the completion of the handshake by the `IEEE80211_AMPDU_TX_OPERATIONAL` action to the **ampdu_action** callback.

Similarly, when the aggregation session is stopped by the peer or something calling *ieee80211_stop_tx_ba_session()*, the driver's **ampdu_action** function will be called with the action `IEEE80211_AMPDU_TX_STOP`. In this case, the call must not fail, and the driver must later call *ieee80211_stop_tx_ba_cb_irqsafe()*. Note that the sta can get destroyed before the BA tear down is complete.

RX A-MPDU aggregation

Aggregation on the RX side requires only implementing the **ampdu_action** callback that is invoked to start/stop any block-ack sessions for RX aggregation.

When RX aggregation is started by the peer, the driver is notified via **ampdu_action** function, with the IEEE80211_AMPDU_RX_START action, and may reject the request in which case a negative response is sent to the peer, if it accepts it a positive response is sent.

While the session is active, the device/driver are required to de-aggregate frames and pass them up one by one to mac80211, which will handle the reorder buffer.

When the aggregation session is stopped again by the peer or ourselves, the driver's **ampdu_action** function will be called with the action `IEEE80211_AMPDU_RX_STOP`. In this case, the call must not fail.

enum **`ieee80211_ampdu_mlme_action`**
A-MPDU actions

Constants

`IEEE80211_AMPDU_RX_START` start RX aggregation

`IEEE80211_AMPDU_RX_STOP` stop RX aggregation

`IEEE80211_AMPDU_TX_START` start TX aggregation

`IEEE80211_AMPDU_TX_STOP_CONT` stop TX aggregation but continue transmitting queued packets, now unaggregated. After all packets are transmitted the driver has to call *ieee80211_stop_tx_ba_cb_irqsafe()*.

`IEEE80211_AMPDU_TX_STOP_FLUSH` stop TX aggregation and flush all packets, called when the station is removed. There's no need or reason to call *ieee80211_stop_tx_ba_cb_irqsafe()* in this case as mac80211 assumes the session is gone and removes the station.

`IEEE80211_AMPDU_TX_STOP_FLUSH_CONT` called when TX aggregation is stopped but the driver hasn't called *ieee80211_stop_tx_ba_cb_irqsafe()* yet and now the connection is dropped and the station will be removed. Drivers should clean up and drop remaining packets when this is called.

IEEE80211_AMPDU_TX_OPERATIONAL TX aggregation has become operational

Description

These flags are used with the `ampdu_action()` callback in *struct ieee80211_ops* to indicate which action is needed.

Note that drivers MUST be able to deal with a TX aggregation session being stopped even before they OK'ed starting it by calling ieee80211_start_tx_ba_cb_irqsafe, because the peer might receive the addBA frame and send a delBA right away!

Spatial Multiplexing Powersave (SMPS)

SMPS (Spatial multiplexing power save) is a mechanism to conserve power in an 802.11n implementation. For details on the mechanism and rationale, please refer to 802.11 (as amended by 802.11n-2009) "11.2.3 SM power save".

The mac80211 implementation is capable of sending action frames to update the AP about the station's SMPS mode, and will instruct the driver to enter the specific mode. It will also announce the requested SMPS mode during the association handshake. Hardware support for this feature is required, and can be indicated by hardware flags.

The default mode will be "automatic", which nl80211/cfg80211 defines to be dynamic SMPS in (regular) powersave, and SMPS turned off otherwise.

To support this feature, the driver must set the appropriate hardware support flags, and handle the SMPS flag to the `config()` operation. It will then with this mechanism be instructed to enter the requested SMPS mode while associated to an HT AP.

void **ieee80211_request_smps**(struct *ieee80211_vif* * *vif*, enum *ieee80211_smps_mode* *smps_mode*)
request SM PS transition

Parameters

struct ieee80211_vif * vif *struct ieee80211_vif* pointer from the add_interface callback.

enum ieee80211_smps_mode smps_mode new SM PS mode

Description

This allows the driver to request an SM PS transition in managed mode. This is useful when the driver has more information than the stack about possible interference, for example by bluetooth.

enum **ieee80211_smps_mode**
spatial multiplexing power save mode

Constants

IEEE80211_SMPS_AUTOMATIC automatic

IEEE80211_SMPS_OFF off

IEEE80211_SMPS_STATIC static

IEEE80211_SMPS_DYNAMIC dynamic

IEEE80211_SMPS_NUM_MODES internal, don't use

TBD

This part of the book describes the rate control algorithm interface and how it relates to mac80211 and drivers.

Rate Control API

TBD

int **ieee80211_start_tx_ba_session**(struct *ieee80211_sta* * *sta*, u16 *tid*, u16 *timeout*)
Start a tx Block Ack session.

Parameters

struct ieee80211_sta * sta the station for which to start a BA session

u16 tid the TID to BA on.

u16 timeout session timeout value (in TUs)

Return

success if addBA request was sent, failure otherwise

Although mac80211/low level driver/user space application can estimate the need to start aggregation on a certain RA/TID, the session level will be managed by the mac80211.

void **ieee80211_start_tx_ba_cb_irqsafe**(struct *ieee80211_vif* * *vif*, const u8 * *ra*, u16 *tid*)
low level driver ready to aggregate.

Parameters

struct ieee80211_vif * vif *struct ieee80211_vif* pointer from the add_interface callback

const u8 * ra receiver address of the BA session recipient.

u16 tid the TID to BA on.

Description

This function must be called by low level driver once it has finished with preparations for the BA session. It can be called from any context.

int **ieee80211_stop_tx_ba_session**(struct *ieee80211_sta* * *sta*, u16 *tid*)
Stop a Block Ack session.

Parameters

struct ieee80211_sta * sta the station whose BA session to stop

u16 tid the TID to stop BA.

Return

negative error if the TID is invalid, or no aggregation active

Although mac80211/low level driver/user space application can estimate the need to stop aggregation on a certain RA/TID, the session level will be managed by the mac80211.

void **ieee80211_stop_tx_ba_cb_irqsafe**(struct *ieee80211_vif* * *vif*, const u8 * *ra*, u16 *tid*)
low level driver ready to stop aggregate.

Parameters

struct ieee80211_vif * vif *struct ieee80211_vif* pointer from the add_interface callback

const u8 * ra receiver address of the BA session recipient.

u16 tid the desired TID to BA on.

Description

This function must be called by low level driver once it has finished with preparations for the BA session tear down. It can be called from any context.

enum **ieee80211_rate_control_changed**
flags to indicate what changed

Constants

IEEE80211_RC_BW_CHANGED The bandwidth that can be used to transmit to this station changed. The actual bandwidth is in the station information – for HT20/40 the IEEE80211_HT_CAP_SUP_WIDTH_20_40 flag changes, for HT and VHT the bandwidth field changes.

IEEE80211_RC_SMPS_CHANGED The SMPS state of the station changed.

IEEE80211_RC_SUPP_RATES_CHANGED The supported rate set of this peer changed (in IBSS mode) due to discovering more information about the peer.

IEEE80211_RC_NSS_CHANGED N_SS (number of spatial streams) was changed by the peer

struct **ieee80211_tx_rate_control**
rate control information for/from RC algo

Definition

```
struct ieee80211_tx_rate_control {
  struct ieee80211_hw *hw;
  struct ieee80211_supported_band *sband;
  struct ieee80211_bss_conf *bss_conf;
  struct sk_buff *skb;
  struct ieee80211_tx_rate reported_rate;
  bool rts, short_preamble;
  u32 rate_idx_mask;
  u8 *rate_idx_mcs_mask;
  bool bss;
};
```

Members

hw The hardware the algorithm is invoked for.

sband The band this frame is being transmitted on.

bss_conf the current BSS configuration

skb the skb that will be transmitted, the control information in it needs to be filled in

reported_rate The rate control algorithm can fill this in to indicate which rate should be reported to userspace as the current rate and used for rate calculations in the mesh network.

rts whether RTS will be used for this frame because it is longer than the RTS threshold

short_preamble whether mac80211 will request short-preamble transmission if the selected rate supports it

rate_idx_mask user-requested (legacy) rate mask

rate_idx_mcs_mask user-requested MCS rate mask (NULL if not in use)

bss whether this frame is sent out in AP or IBSS mode

bool **rate_control_send_low**(struct *ieee80211_sta* * *sta*, void * *priv_sta*, struct *ieee80211_tx_rate_control* * *txrc*)
helper for drivers for management/no-ack frames

Parameters

struct ieee80211_sta * sta *struct ieee80211_sta* pointer to the target destination. Note that this may be null.

void * priv_sta private rate control structure. This may be null.

struct ieee80211_tx_rate_control * txrc rate control information we sholud populate for mac80211.

Description

Rate control algorithms that agree to use the lowest rate to send management frames and NO_ACK data with the respective hw retries should use this in the beginning of their mac80211 get_rate callback. If true is returned the rate control can simply return. If false is returned we guarantee that sta and sta and priv_sta is not null.

Rate control algorithms wishing to do more intelligent selection of rate for multicast/broadcast frames may choose to not use this.

TBD

This part of the book describes mac80211 internals.

Key handling

Key handling basics

Key handling in mac80211 is done based on per-interface (sub_if_data) keys and per-station keys. Since each station belongs to an interface, each station key also belongs to that interface.

Hardware acceleration is done on a best-effort basis for algorithms that are implemented in software, for each key the hardware is asked to enable that key for offloading but if it cannot do that the key is simply kept for software encryption (unless it is for an algorithm that isn't implemented in software). There is currently no way of knowing whether a key is handled in SW or HW except by looking into debugfs.

All key management is internally protected by a mutex. Within all other parts of mac80211, key references are, just as STA structure references, protected by RCU. Note, however, that some things are unprotected, namely the key->sta dereferences within the hardware acceleration functions. This means that `sta_info_destroy()` must remove the key which waits for an RCU grace period.

MORE TBD

TBD

Receive processing

TBD

Transmit processing

TBD

Station info handling

Programming information

struct **`sta_info`**
  STA information

Definition

```
struct sta_info {
  struct list_head list, free_list;
  struct rcu_head rcu_head;
  struct rhlist_head hash_node;
  u8 addr[ETH_ALEN];
  struct ieee80211_local *local;
  struct ieee80211_sub_if_data *sdata;
  struct ieee80211_key __rcu *gtk[NUM_DEFAULT_KEYS + NUM_DEFAULT_MGMT_KEYS];
  struct ieee80211_key __rcu *ptk[NUM_DEFAULT_KEYS];
  u8 ptk_idx;
  struct rate_control_ref *rate_ctrl;
  void *rate_ctrl_priv;
  spinlock_t rate_ctrl_lock;
  spinlock_t lock;
  struct ieee80211_fast_tx __rcu *fast_tx;
  struct ieee80211_fast_rx __rcu *fast_rx;
  struct ieee80211_sta_rx_stats __percpu *pcpu_rx_stats;
#ifdef CONFIG_MAC80211_MESH;
  struct mesh_sta *mesh;
#endif;
  struct work_struct drv_deliver_wk;
  u16 listen_interval;
  bool dead;
  bool removed;
  bool uploaded;
  enum ieee80211_sta_state sta_state;
  unsigned long _flags;
  spinlock_t ps_lock;
  struct sk_buff_head ps_tx_buf[IEEE80211_NUM_ACS];
  struct sk_buff_head tx_filtered[IEEE80211_NUM_ACS];
  unsigned long driver_buffered_tids;
  unsigned long txq_buffered_tids;
  long last_connected;
  struct ieee80211_sta_rx_stats rx_stats;
  struct {
    struct ewma_signal signal;
    struct ewma_signal chain_signal[IEEE80211_MAX_CHAINS];
  } rx_stats_avg;
  __le16 last_seq_ctrl[IEEE80211_NUM_TIDS + 1];
  struct {
    unsigned long filtered;
    unsigned long retry_failed, retry_count;
    unsigned int lost_packets;
    unsigned long last_tdls_pkt_time;
    u64 msdu_retries[IEEE80211_NUM_TIDS + 1];
    u64 msdu_failed[IEEE80211_NUM_TIDS + 1];
    unsigned long last_ack;
  } status_stats;
  struct {
    u64 packets[IEEE80211_NUM_ACS];
    u64 bytes[IEEE80211_NUM_ACS];
    struct ieee80211_tx_rate last_rate;
    u64 msdu[IEEE80211_NUM_TIDS + 1];
  } tx_stats;
  u16 tid_seq[IEEE80211_QOS_CTL_TID_MASK + 1];
  struct sta_ampdu_mlme ampdu_mlme;
#ifdef CONFIG_MAC80211_DEBUGFS;
  struct dentry *debugfs_dir;
#endif;
  enum ieee80211_sta_rx_bandwidth cur_max_bandwidth;
  enum ieee80211_smps_mode known_smps_mode;
  const struct ieee80211_cipher_scheme *cipher_scheme;
  struct codel_params cparams;
  u8 reserved_tid;
  struct cfg80211_chan_def tdls_chandef;
  struct ieee80211_sta sta;
};
```

Members

list global linked list entry

free_list list entry for keeping track of stations to free

rcu_head RCU head used for freeing this station struct

hash_node hash node for rhashtable

addr station's MAC address - duplicated from public part to let the hash table work with just a single cacheline

local pointer to the global information

sdata virtual interface this station belongs to

gtk group keys negotiated with this station, if any

ptk peer keys negotiated with this station, if any

ptk_idx last installed peer key index

rate_ctrl rate control algorithm reference

rate_ctrl_priv rate control private per-STA pointer

rate_ctrl_lock spinlock used to protect rate control data (data inside the algorithm, so serializes calls there)

lock used for locking all fields that require locking, see comments in the header file.

fast_tx TX fastpath information

fast_rx RX fastpath information

pcpu_rx_stats per-CPU RX statistics, assigned only if the driver needs this (by advertising the USES_RSS hw flag)

mesh mesh STA information

drv_deliver_wk used for delivering frames after driver PS unblocking

listen_interval listen interval of this station, when we're acting as AP

dead set to true when sta is unlinked

removed set to true when sta is being removed from sta_list

uploaded set to true when sta is uploaded to the driver

sta_state duplicates information about station state (for debug)

_flags STA flags, see *enum ieee80211_sta_info_flags*, do not use directly

ps_lock used for powersave (when mac80211 is the AP) related locking

ps_tx_buf buffers (per AC) of frames to transmit to this station when it leaves power saving state or polls

tx_filtered buffers (per AC) of frames we already tried to transmit but were filtered by hardware due to STA having entered power saving state, these are also delivered to the station when it leaves powersave or polls for frames

driver_buffered_tids bitmap of TIDs the driver has data buffered on

txq_buffered_tids bitmap of TIDs that mac80211 has txq data buffered on

last_connected time (in seconds) when a station got connected

rx_stats RX statistics

last_seq_ctrl last received seq/frag number from this STA (per TID plus one for non-QoS frames)

status_stats TX status statistics

tx_stats TX statistics

tid_seq per-TID sequence numbers for sending to this STA

ampdu_mlme A-MPDU state machine state

debugfs_dir debug filesystem directory dentry

cur_max_bandwidth maximum bandwidth to use for TX to the station, taken from HT/VHT capabilities or VHT operating mode notification

known_smps_mode the smps_mode the client thinks we are in. Relevant for AP only.

cipher_scheme optional cipher scheme for this station

cparams CoDel parameters for this station.

reserved_tid reserved TID (if any, otherwise IEEE80211_TID_UNRESERVED)

tdls_chandef a TDLS peer can have a wider chandef that is compatible to the BSS one.

sta station information we share with the driver

Description

This structure collects information about a station that mac80211 is communicating with.

enum **ieee80211_sta_info_flags**
Stations flags

Constants

WLAN_STA_AUTH Station is authenticated.

WLAN_STA_ASSOC Station is associated.

WLAN_STA_PS_STA Station is in power-save mode

WLAN_STA_AUTHORIZED Station is authorized to send/receive traffic. This bit is always checked so needs to be enabled for all stations when virtual port control is not in use.

WLAN_STA_SHORT_PREAMBLE Station is capable of receiving short-preamble frames.

WLAN_STA_WDS Station is one of our WDS peers.

WLAN_STA_CLEAR_PS_FILT Clear PS filter in hardware (using the IEEE80211_TX_CTL_CLEAR_PS_FILT control flag) when the next frame to this station is transmitted.

WLAN_STA_MFP Management frame protection is used with this STA.

WLAN_STA_BLOCK_BA Used to deny ADDBA requests (both TX and RX) during suspend/resume and station removal.

WLAN_STA_PS_DRIVER driver requires keeping this station in power-save mode logically to flush frames that might still be in the queues

WLAN_STA_PSPOLL Station sent PS-poll while driver was keeping station in power-save mode, reply when the driver unblocks.

WLAN_STA_TDLS_PEER Station is a TDLS peer.

WLAN_STA_TDLS_PEER_AUTH This TDLS peer is authorized to send direct packets. This means the link is enabled.

WLAN_STA_TDLS_INITIATOR We are the initiator of the TDLS link with this station.

WLAN_STA_TDLS_CHAN_SWITCH This TDLS peer supports TDLS channel-switching

WLAN_STA_TDLS_OFF_CHANNEL The local STA is currently off-channel with this TDLS peer

WLAN_STA_TDLS_WIDER_BW This TDLS peer supports working on a wider bw on the BSS base channel.

WLAN_STA_UAPSD Station requested unscheduled SP while driver was keeping station in power-save mode, reply when the driver unblocks the station.

WLAN_STA_SP Station is in a service period, so don't try to reply to other uAPSD trigger frames or PS-Poll.

WLAN_STA_4ADDR_EVENT 4-addr event was already sent for this frame.

WLAN_STA_INSERTED This station is inserted into the hash table.

WLAN_STA_RATE_CONTROL rate control was initialized for this station.

WLAN_STA_TOFFSET_KNOWN toffset calculated for this station is valid.

WLAN_STA_MPSP_OWNER local STA is owner of a mesh Peer Service Period.

WLAN_STA_MPSP_RECIPIENT local STA is recipient of a MPSP.

WLAN_STA_PS_DELIVER station woke up, but we're still blocking TX until pending frames are delivered

`NUM_WLAN_STA_FLAGS` number of defined flags

Description

These flags are used with *`struct sta_info`*'s **flags** member, but only indirectly with `set_sta_flag()` and friends.

STA information lifetime rules

STA info structures (*`struct sta_info`*) are managed in a hash table for faster lookup and a list for iteration. They are managed using RCU, i.e. access to the list and hash table is protected by RCU.

Upon allocating a STA info structure with `sta_info_alloc()`, the caller owns that structure. It must then insert it into the hash table using either `sta_info_insert()` or `sta_info_insert_rcu()`; only in the latter case (which acquires an rcu read section but must not be called from within one) will the pointer still be valid after the call. Note that the caller may not do much with the STA info before inserting it, in particular, it may not start any mesh peer link management or add encryption keys.

When the insertion fails (`sta_info_insert()`) returns non-zero), the structure will have been freed by `sta_info_insert()`!

Station entries are added by mac80211 when you establish a link with a peer. This means different things for the different type of interfaces we support. For a regular station this mean we add the AP sta when we receive an association response from the AP. For IBSS this occurs when get to know about a peer on the same IBSS. For WDS we add the sta for the peer immediately upon device open. When using AP mode we add stations for each respective station upon request from userspace through nl80211.

In order to remove a STA info structure, various sta_info_destroy_*() calls are available.

There is no concept of ownership on a STA entry, each structure is owned by the global hash table/list until it is removed. All users of the structure need to be RCU protected so that the structure won't be freed before they are done using it.

Aggregation

struct **`sta_ampdu_mlme`**
STA aggregation information.

Definition

```
struct sta_ampdu_mlme {
  struct mutex mtx;
  struct tid_ampdu_rx __rcu *tid_rx[IEEE80211_NUM_TIDS];
  u8 tid_rx_token[IEEE80211_NUM_TIDS];
  unsigned long tid_rx_timer_expired[BITS_TO_LONGS(IEEE80211_NUM_TIDS)];
  unsigned long tid_rx_stop_requested[BITS_TO_LONGS(IEEE80211_NUM_TIDS)];
  unsigned long tid_rx_manage_offl[BITS_TO_LONGS(2 * IEEE80211_NUM_TIDS)];
  unsigned long agg_session_valid[BITS_TO_LONGS(IEEE80211_NUM_TIDS)];
  unsigned long unexpected_agg[BITS_TO_LONGS(IEEE80211_NUM_TIDS)];
  struct work_struct work;
  struct tid_ampdu_tx __rcu *tid_tx[IEEE80211_NUM_TIDS];
  struct tid_ampdu_tx *tid_start_tx[IEEE80211_NUM_TIDS];
  unsigned long last_addba_req_time[IEEE80211_NUM_TIDS];
  u8 addba_req_num[IEEE80211_NUM_TIDS];
  u8 dialog_token_allocator;
};
```

Members

`mtx` mutex to protect all TX data (except non-NULL assignments to tid_tx[idx], which are protected by the sta spinlock) tid_start_tx is also protected by sta->lock.

`tid_rx` aggregation info for Rx per TID – RCU protected

tid_rx_token dialog tokens for valid aggregation sessions

tid_rx_timer_expired bitmap indicating on which TIDs the RX timer expired until the work for it runs

tid_rx_stop_requested bitmap indicating which BA sessions per TID the driver requested to close until the work for it runs

tid_rx_manage_offl bitmap indicating which BA sessions were requested to be treated as started/stopped due to offloading

agg_session_valid bitmap indicating which TID has a rx BA session open on

unexpected_agg bitmap indicating which TID already sent a delBA due to unexpected aggregation related frames outside a session

work work struct for starting/stopping aggregation

tid_tx aggregation info for Tx per TID

tid_start_tx sessions where start was requested

last_addba_req_time timestamp of the last addBA request.

addba_req_num number of times addBA request has been sent.

dialog_token_allocator dialog token enumerator for each new session;

struct **tid_ampdu_tx**
TID aggregation information (Tx).

Definition

```
struct tid_ampdu_tx {
  struct rcu_head rcu_head;
  struct timer_list session_timer;
  struct timer_list addba_resp_timer;
  struct sk_buff_head pending;
  struct sta_info *sta;
  unsigned long state;
  unsigned long last_tx;
  u16 timeout;
  u8 dialog_token;
  u8 stop_initiator;
  bool tx_stop;
  u8 buf_size;
  u16 failed_bar_ssn;
  bool bar_pending;
  bool amsdu;
  u8 tid;
};
```

Members

rcu_head rcu head for freeing structure

session_timer check if we keep Tx-ing on the TID (by timeout value)

addba_resp_timer timer for peer’s response to addba request

pending pending frames queue – use sta’s spinlock to protect

sta station we are attached to

state session state (see above)

last_tx jiffies of last tx activity

timeout session timeout value to be filled in ADDBA requests

dialog_token dialog token for aggregation session

stop_initiator initiator of a session stop

tx_stop TX DelBA frame when stopping

buf_size reorder buffer size at receiver

failed_bar_ssn ssn of the last failed BAR tx attempt

bar_pending BAR needs to be re-sent

amsdu support A-MSDU withing A-MDPU

tid TID number

Description

This structure's lifetime is managed by RCU, assignments to the array holding it must hold the aggregation mutex.

The TX path can access it under RCU lock-free if, and only if, the state has the flag HT_AGG_STATE_OPERATIONAL set. Otherwise, the TX path must also acquire the spinlock and re-check the state, see comments in the tx code touching it.

struct **tid_ampdu_rx**
TID aggregation information (Rx).

Definition

```
struct tid_ampdu_rx {
  struct rcu_head rcu_head;
  spinlock_t reorder_lock;
  u64 reorder_buf_filtered;
  struct sk_buff_head *reorder_buf;
  unsigned long *reorder_time;
  struct sta_info *sta;
  struct timer_list session_timer;
  struct timer_list reorder_timer;
  unsigned long last_rx;
  u16 head_seq_num;
  u16 stored_mpdu_num;
  u16 ssn;
  u16 buf_size;
  u16 timeout;
  u8 tid;
  u8 auto_seq:1,removed:1, started:1;
};
```

Members

rcu_head RCU head used for freeing this struct

reorder_lock serializes access to reorder buffer, see below.

reorder_buf_filtered bitmap indicating where there are filtered frames in the reorder buffer that should be ignored when releasing frames

reorder_buf buffer to reorder incoming aggregated MPDUs. An MPDU may be an A-MSDU with individually reported subframes.

reorder_time jiffies when skb was added

sta station we are attached to

session_timer check if peer keeps Tx-ing on the TID (by timeout value)

reorder_timer releases expired frames from the reorder buffer.

last_rx jiffies of last rx activity

head_seq_num head sequence number in reordering buffer.

stored_mpdu_num number of MPDUs in reordering buffer

ssn Starting Sequence Number expected to be aggregated.

buf_size buffer size for incoming A-MPDUs

timeout reset timer value (in TUs).

tid TID number

auto_seq used for offloaded BA sessions to automatically pick head_seq_and and ssn.

removed this session is removed (but might have been found due to RCU)

started this session has started (head ssn or higher was received)

Description

This structure's lifetime is managed by RCU, assignments to the array holding it must hold the aggregation mutex.

The **reorder_lock** is used to protect the members of this struct, except for **timeout**, **buf_size** and **dialog_token**, which are constant across the lifetime of the struct (the dialog token being used only for debugging).

Synchronisation

TBD

Locking, lots of RCU

CHAPTER

THIRTYONE

THE USERSPACE I/O HOWTO

Author Hans-Jürgen Koch Linux developer, Linutronix

Date 2006-12-11

About this document

Translations

If you know of any translations for this document, or you are interested in translating it, please email me hjk@hansjkoch.de.

Preface

For many types of devices, creating a Linux kernel driver is overkill. All that is really needed is some way to handle an interrupt and provide access to the memory space of the device. The logic of controlling the device does not necessarily have to be within the kernel, as the device does not need to take advantage of any of other resources that the kernel provides. One such common class of devices that are like this are for industrial I/O cards.

To address this situation, the userspace I/O system (UIO) was designed. For typical industrial I/O cards, only a very small kernel module is needed. The main part of the driver will run in user space. This simplifies development and reduces the risk of serious bugs within a kernel module.

Please note that UIO is not an universal driver interface. Devices that are already handled well by other kernel subsystems (like networking or serial or USB) are no candidates for an UIO driver. Hardware that is ideally suited for an UIO driver fulfills all of the following:

- The device has memory that can be mapped. The device can be controlled completely by writing to this memory.
- The device usually generates interrupts.
- The device does not fit into one of the standard kernel subsystems.

Acknowledgments

I'd like to thank Thomas Gleixner and Benedikt Spranger of Linutronix, who have not only written most of the UIO code, but also helped greatly writing this HOWTO by giving me all kinds of background information.

Feedback

Find something wrong with this document? (Or perhaps something right?) I would love to hear from you. Please email me at hjk@hansjkoch.de.

About UIO

If you use UIO for your card's driver, here's what you get:

- only one small kernel module to write and maintain.
- develop the main part of your driver in user space, with all the tools and libraries you're used to.
- bugs in your driver won't crash the kernel.
- updates of your driver can take place without recompiling the kernel.

How UIO works

Each UIO device is accessed through a device file and several sysfs attribute files. The device file will be called `/dev/uio0` for the first device, and `/dev/uio1`, `/dev/uio2` and so on for subsequent devices.

`/dev/uioX` is used to access the address space of the card. Just use `mmap()` to access registers or RAM locations of your card.

Interrupts are handled by reading from `/dev/uioX`. A blocking `read()` from `/dev/uioX` will return as soon as an interrupt occurs. You can also use `select()` on `/dev/uioX` to wait for an interrupt. The integer value read from `/dev/uioX` represents the total interrupt count. You can use this number to figure out if you missed some interrupts.

For some hardware that has more than one interrupt source internally, but not separate IRQ mask and status registers, there might be situations where userspace cannot determine what the interrupt source was if the kernel handler disables them by writing to the chip's IRQ register. In such a case, the kernel has to disable the IRQ completely to leave the chip's register untouched. Now the userspace part can determine the cause of the interrupt, but it cannot re-enable interrupts. Another cornercase is chips where re-enabling interrupts is a read-modify-write operation to a combined IRQ status/acknowledge register. This would be racy if a new interrupt occurred simultaneously.

To address these problems, UIO also implements a write() function. It is normally not used and can be ignored for hardware that has only a single interrupt source or has separate IRQ mask and status registers. If you need it, however, a write to `/dev/uioX` will call the `irqcontrol()` function implemented by the driver. You have to write a 32-bit value that is usually either 0 or 1 to disable or enable interrupts. If a driver does not implement `irqcontrol()`, `write()` will return with `-ENOSYS`.

To handle interrupts properly, your custom kernel module can provide its own interrupt handler. It will automatically be called by the built-in handler.

For cards that don't generate interrupts but need to be polled, there is the possibility to set up a timer that triggers the interrupt handler at configurable time intervals. This interrupt simulation is done by calling *`uio_event_notify()`* from the timer's event handler.

Each driver provides attributes that are used to read or write variables. These attributes are accessible through sysfs files. A custom kernel driver module can add its own attributes to the device owned by the uio driver, but not added to the UIO device itself at this time. This might change in the future if it would be found to be useful.

The following standard attributes are provided by the UIO framework:

- `name`: The name of your device. It is recommended to use the name of your kernel module for this.
- `version`: A version string defined by your driver. This allows the user space part of your driver to deal with different versions of the kernel module.
- `event`: The total number of interrupts handled by the driver since the last time the device node was read.

These attributes appear under the `/sys/class/uio/uioX` directory. Please note that this directory might be a symlink, and not a real directory. Any userspace code that accesses it must be able to handle this.

Each UIO device can make one or more memory regions available for memory mapping. This is necessary because some industrial I/O cards require access to more than one PCI memory region in a driver.

Each mapping has its own directory in sysfs, the first mapping appears as `/sys/class/uio/uioX/maps/map0/`. Subsequent mappings create directories `map1/`, `map2/`, and so on. These directories will only appear if the size of the mapping is not 0.

Each `mapX/` directory contains four read-only files that show attributes of the memory:

- `name`: A string identifier for this mapping. This is optional, the string can be empty. Drivers can set this to make it easier for userspace to find the correct mapping.
- `addr`: The address of memory that can be mapped.
- `size`: The size, in bytes, of the memory pointed to by addr.
- `offset`: The offset, in bytes, that has to be added to the pointer returned by `mmap()` to get to the actual device memory. This is important if the device's memory is not page aligned. Remember that pointers returned by `mmap()` are always page aligned, so it is good style to always add this offset.

From userspace, the different mappings are distinguished by adjusting the `offset` parameter of the `mmap()` call. To map the memory of mapping N, you have to use N times the page size as your offset:

```
offset = N * getpagesize();
```

Sometimes there is hardware with memory-like regions that can not be mapped with the technique described here, but there are still ways to access them from userspace. The most common example are x86 ioports. On x86 systems, userspace can access these ioports using `ioperm()`, `iopl()`, `inb()`, `outb()`, and similar functions.

Since these ioport regions can not be mapped, they will not appear under `/sys/class/uio/uioX/maps/` like the normal memory described above. Without information about the port regions a hardware has to offer, it becomes difficult for the userspace part of the driver to find out which ports belong to which UIO device.

To address this situation, the new directory `/sys/class/uio/uioX/portio/` was added. It only exists if the driver wants to pass information about one or more port regions to userspace. If that is the case, subdirectories named `port0`, `port1`, and so on, will appear underneath `/sys/class/uio/uioX/portio/`.

Each `portX/` directory contains four read-only files that show name, start, size, and type of the port region:

- `name`: A string identifier for this port region. The string is optional and can be empty. Drivers can set it to make it easier for userspace to find a certain port region.
- `start`: The first port of this region.
- `size`: The number of ports in this region.
- `porttype`: A string describing the type of port.

Writing your own kernel module

Please have a look at `uio_cif.c` as an example. The following paragraphs explain the different sections of this file.

struct uio_info

This structure tells the framework the details of your driver, Some of the members are required, others are optional.

- `const char *name`: Required. The name of your driver as it will appear in sysfs. I recommend using the name of your module for this.
- `const char *version`: Required. This string appears in `/sys/class/uio/uioX/version`.

- `struct uio_mem mem[ MAX_UIO_MAPS ]`: Required if you have memory that can be mapped with `mmap()`. For each mapping you need to fill one of the `uio_mem` structures. See the description below for details.
- `struct uio_port port[ MAX_UIO_PORTS_REGIONS ]`: Required if you want to pass information about ioports to userspace. For each port region you need to fill one of the `uio_port` structures. See the description below for details.
- `long irq`: Required. If your hardware generates an interrupt, it's your modules task to determine the irq number during initialization. If you don't have a hardware generated interrupt but want to trigger the interrupt handler in some other way, set `irq` to `UIO_IRQ_CUSTOM`. If you had no interrupt at all, you could set `irq` to `UIO_IRQ_NONE`, though this rarely makes sense.
- `unsigned long irq_flags`: Required if you've set `irq` to a hardware interrupt number. The flags given here will be used in the call to `request_irq()`.
- `int (*mmap)(struct uio_info *info, struct vm_area_struct *vma)`: Optional. If you need a special `mmap()` function, you can set it here. If this pointer is not NULL, your `mmap()` will be called instead of the built-in one.
- `int (*open)(struct uio_info *info, struct inode *inode)`: Optional. You might want to have your own `open()`, e.g. to enable interrupts only when your device is actually used.
- `int (*release)(struct uio_info *info, struct inode *inode)`: Optional. If you define your own `open()`, you will probably also want a custom `release()` function.
- `int (*irqcontrol)(struct uio_info *info, s32 irq_on)`: Optional. If you need to be able to enable or disable interrupts from userspace by writing to `/dev/uioX`, you can implement this function. The parameter `irq_on` will be 0 to disable interrupts and 1 to enable them.

Usually, your device will have one or more memory regions that can be mapped to user space. For each region, you have to set up a `struct uio_mem` in the `mem[]` array. Here's a description of the fields of `struct uio_mem`:

- `const char *name`: Optional. Set this to help identify the memory region, it will show up in the corresponding sysfs node.
- `int memtype`: Required if the mapping is used. Set this to `UIO_MEM_PHYS` if you you have physical memory on your card to be mapped. Use `UIO_MEM_LOGICAL` for logical memory (e.g. allocated with `kmalloc()`). There's also `UIO_MEM_VIRTUAL` for virtual memory.
- `phys_addr_t addr`: Required if the mapping is used. Fill in the address of your memory block. This address is the one that appears in sysfs.
- `resource_size_t size`: Fill in the size of the memory block that `addr` points to. If `size` is zero, the mapping is considered unused. Note that you *must* initialize `size` with zero for all unused mappings.
- `void *internal_addr`: If you have to access this memory region from within your kernel module, you will want to map it internally by using something like *`ioremap()`*. Addresses returned by this function cannot be mapped to user space, so you must not store it in `addr`. Use `internal_addr` instead to remember such an address.

Please do not touch the `map` element of `struct uio_mem`! It is used by the UIO framework to set up sysfs files for this mapping. Simply leave it alone.

Sometimes, your device can have one or more port regions which can not be mapped to userspace. But if there are other possibilities for userspace to access these ports, it makes sense to make information about the ports available in sysfs. For each region, you have to set up a `struct uio_port` in the `port[]` array. Here's a description of the fields of `struct uio_port`:

- `char *porttype`: Required. Set this to one of the predefined constants. Use `UIO_PORT_X86` for the ioports found in x86 architectures.
- `unsigned long start`: Required if the port region is used. Fill in the number of the first port of this region.

- `unsigned long size`: Fill in the number of ports in this region. If `size` is zero, the region is considered unused. Note that you *must* initialize `size` with zero for all unused regions.

Please do not touch the `portio` element of `struct uio_port`! It is used internally by the UIO framework to set up sysfs files for this region. Simply leave it alone.

Adding an interrupt handler

What you need to do in your interrupt handler depends on your hardware and on how you want to handle it. You should try to keep the amount of code in your kernel interrupt handler low. If your hardware requires no action that you *have* to perform after each interrupt, then your handler can be empty.

If, on the other hand, your hardware *needs* some action to be performed after each interrupt, then you *must* do it in your kernel module. Note that you cannot rely on the userspace part of your driver. Your userspace program can terminate at any time, possibly leaving your hardware in a state where proper interrupt handling is still required.

There might also be applications where you want to read data from your hardware at each interrupt and buffer it in a piece of kernel memory you've allocated for that purpose. With this technique you could avoid loss of data if your userspace program misses an interrupt.

A note on shared interrupts: Your driver should support interrupt sharing whenever this is possible. It is possible if and only if your driver can detect whether your hardware has triggered the interrupt or not. This is usually done by looking at an interrupt status register. If your driver sees that the IRQ bit is actually set, it will perform its actions, and the handler returns IRQ_HANDLED. If the driver detects that it was not your hardware that caused the interrupt, it will do nothing and return IRQ_NONE, allowing the kernel to call the next possible interrupt handler.

If you decide not to support shared interrupts, your card won't work in computers with no free interrupts. As this frequently happens on the PC platform, you can save yourself a lot of trouble by supporting interrupt sharing.

Using uio_pdrv for platform devices

In many cases, UIO drivers for platform devices can be handled in a generic way. In the same place where you define your `struct platform_device`, you simply also implement your interrupt handler and fill your `struct uio_info`. A pointer to this `struct uio_info` is then used as `platform_data` for your platform device.

You also need to set up an array of `struct resource` containing addresses and sizes of your memory mappings. This information is passed to the driver using the `.resource` and `.num_resources` elements of `struct platform_device`.

You now have to set the `.name` element of `struct platform_device` to `"uio_pdrv"` to use the generic UIO platform device driver. This driver will fill the `mem[]` array according to the resources given, and register the device.

The advantage of this approach is that you only have to edit a file you need to edit anyway. You do not have to create an extra driver.

Using uio_pdrv_genirq for platform devices

Especially in embedded devices, you frequently find chips where the irq pin is tied to its own dedicated interrupt line. In such cases, where you can be really sure the interrupt is not shared, we can take the concept of `uio_pdrv` one step further and use a generic interrupt handler. That's what `uio_pdrv_genirq` does.

The setup for this driver is the same as described above for `uio_pdrv`, except that you do not implement an interrupt handler. The `.handler` element of `struct uio_info` must remain `NULL`. The `.irq_flags` element must not contain `IRQF_SHARED`.

You will set the .name element of `struct platform_device` to "`uio_pdrv_genirq`" to use this driver.

The generic interrupt handler of `uio_pdrv_genirq` will simply disable the interrupt line using `disable_irq_nosync()`. After doing its work, userspace can reenable the interrupt by writing 0x00000001 to the UIO device file. The driver already implements an `irq_control()` to make this possible, you must not implement your own.

Using `uio_pdrv_genirq` not only saves a few lines of interrupt handler code. You also do not need to know anything about the chip's internal registers to create the kernel part of the driver. All you need to know is the irq number of the pin the chip is connected to.

Using uio_dmem_genirq for platform devices

In addition to statically allocated memory ranges, they may also be a desire to use dynamically allocated regions in a user space driver. In particular, being able to access memory made available through the dma-mapping API, may be particularly useful. The `uio_dmem_genirq` driver provides a way to accomplish this.

This driver is used in a similar manner to the "`uio_pdrv_genirq`" driver with respect to interrupt configuration and handling.

Set the .name element of `struct platform_device` to "`uio_dmem_genirq`" to use this driver.

When using this driver, fill in the `.platform_data` element of `struct platform_device`, which is of type `struct uio_dmem_genirq_pdata` and which contains the following elements:

- `struct uio_info uioinfo`: The same structure used as the `uio_pdrv_genirq` platform data
- `unsigned int *dynamic_region_sizes`: Pointer to list of sizes of dynamic memory regions to be mapped into user space.
- `unsigned int num_dynamic_regions`: Number of elements in `dynamic_region_sizes` array.

The dynamic regions defined in the platform data will be appended to the ‘‘ mem[] ‘‘ array after the platform device resources, which implies that the total number of static and dynamic memory regions cannot exceed `MAX_UIO_MAPS`.

The dynamic memory regions will be allocated when the UIO device file, `/dev/uioX` is opened. Similar to static memory resources, the memory region information for dynamic regions is then visible via sysfs at `/sys/class/uio/uioX/maps/mapY/*`. The dynamic memory regions will be freed when the UIO device file is closed. When no processes are holding the device file open, the address returned to userspace is ~0.

Writing a driver in userspace

Once you have a working kernel module for your hardware, you can write the userspace part of your driver. You don't need any special libraries, your driver can be written in any reasonable language, you can use floating point numbers and so on. In short, you can use all the tools and libraries you'd normally use for writing a userspace application.

Getting information about your UIO device

Information about all UIO devices is available in sysfs. The first thing you should do in your driver is check `name` and `version` to make sure your talking to the right device and that its kernel driver has the version you expect.

You should also make sure that the memory mapping you need exists and has the size you expect.

There is a tool called `lsuio` that lists UIO devices and their attributes. It is available here:

http://www.osadl.org/projects/downloads/UIO/user/

With `lsuio` you can quickly check if your kernel module is loaded and which attributes it exports. Have a look at the manpage for details.

The source code of `lsuio` can serve as an example for getting information about an UIO device. The file `uio_helper.c` contains a lot of functions you could use in your userspace driver code.

mmap() device memory

After you made sure you've got the right device with the memory mappings you need, all you have to do is to call `mmap()` to map the device's memory to userspace.

The parameter `offset` of the `mmap()` call has a special meaning for UIO devices: It is used to select which mapping of your device you want to map. To map the memory of mapping N, you have to use N times the page size as your offset:

```
offset = N * getpagesize();
```

N starts from zero, so if you've got only one memory range to map, set `offset = 0`. A drawback of this technique is that memory is always mapped beginning with its start address.

Waiting for interrupts

After you successfully mapped your devices memory, you can access it like an ordinary array. Usually, you will perform some initialization. After that, your hardware starts working and will generate an interrupt as soon as it's finished, has some data available, or needs your attention because an error occurred.

`/dev/uioX` is a read-only file. A `read()` will always block until an interrupt occurs. There is only one legal value for the `count` parameter of `read()`, and that is the size of a signed 32 bit integer (4). Any other value for `count` causes `read()` to fail. The signed 32 bit integer read is the interrupt count of your device. If the value is one more than the value you read the last time, everything is OK. If the difference is greater than one, you missed interrupts.

You can also use `select()` on `/dev/uioX`.

Generic PCI UIO driver

The generic driver is a kernel module named uio_pci_generic. It can work with any device compliant to PCI 2.3 (circa 2002) and any compliant PCI Express device. Using this, you only need to write the userspace driver, removing the need to write a hardware-specific kernel module.

Making the driver recognize the device

Since the driver does not declare any device ids, it will not get loaded automatically and will not automatically bind to any devices, you must load it and allocate id to the driver yourself. For example:

```
modprobe uio_pci_generic
echo "8086 10f5" > /sys/bus/pci/drivers/uio_pci_generic/new_id
```

If there already is a hardware specific kernel driver for your device, the generic driver still won't bind to it, in this case if you want to use the generic driver (why would you?) you'll have to manually unbind the hardware specific driver and bind the generic driver, like this:

```
echo -n 0000:00:19.0 > /sys/bus/pci/drivers/e1000e/unbind
echo -n 0000:00:19.0 > /sys/bus/pci/drivers/uio_pci_generic/bind
```

You can verify that the device has been bound to the driver by looking for it in sysfs, for example like the following:

```
ls -l /sys/bus/pci/devices/0000:00:19.0/driver
```

Which if successful should print:

```
.../0000:00:19.0/driver -> ../../../bus/pci/drivers/uio_pci_generic
```

Note that the generic driver will not bind to old PCI 2.2 devices. If binding the device failed, run the following command:

```
dmesg
```

and look in the output for failure reasons.

Things to know about uio_pci_generic

Interrupts are handled using the Interrupt Disable bit in the PCI command register and Interrupt Status bit in the PCI status register. All devices compliant to PCI 2.3 (circa 2002) and all compliant PCI Express devices should support these bits. uio_pci_generic detects this support, and won't bind to devices which do not support the Interrupt Disable Bit in the command register.

On each interrupt, uio_pci_generic sets the Interrupt Disable bit. This prevents the device from generating further interrupts until the bit is cleared. The userspace driver should clear this bit before blocking and waiting for more interrupts.

Writing userspace driver using uio_pci_generic

Userspace driver can use pci sysfs interface, or the libpci library that wraps it, to talk to the device and to re-enable interrupts by writing to the command register.

Example code using uio_pci_generic

Here is some sample userspace driver code using uio_pci_generic:

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>

int main()
{
    int uiofd;
    int configfd;
    int err;
    int i;
    unsigned icount;
    unsigned char command_high;

    uiofd = open("/dev/uio0", O_RDONLY);
    if (uiofd < 0) {
        perror("uio open:");
        return errno;
    }
    configfd = open("/sys/class/uio/uio0/device/config", O_RDWR);
    if (configfd < 0) {
        perror("config open:");
        return errno;
    }

    /* Read and cache command value */
    err = pread(configfd, &command_high, 1, 5);
    if (err != 1) {
        perror("command config read:");
        return errno;
    }
    command_high &= ~0x4;

    for(i = 0;; ++i) {
        /* Print out a message, for debugging. */
        if (i == 0)
            fprintf(stderr, "Started uio test driver.\n");
        else
            fprintf(stderr, "Interrupts: %d\n", icount);

        /****************************************/
        /* Here we got an interrupt from the
           device. Do something to it. */
        /****************************************/

        /* Re-enable interrupts. */
        err = pwrite(configfd, &command_high, 1, 5);
        if (err != 1) {
            perror("config write:");
            break;
        }

        /* Wait for next interrupt. */
        err = read(uiofd, &icount, 4);
        if (err != 4) {
            perror("uio read:");
            break;
        }

    }
    return errno;
}
```

Generic Hyper-V UIO driver

The generic driver is a kernel module named uio_hv_generic. It supports devices on the Hyper-V VMBus similar to uio_pci_generic on PCI bus.

Making the driver recognize the device

Since the driver does not declare any device GUID's, it will not get loaded automatically and will not automatically bind to any devices, you must load it and allocate id to the driver yourself. For example, to use the network device class GUID:

```
modprobe uio_hv_generic
echo "f8615163-df3e-46c5-913f-f2d2f965ed0e" > /sys/bus/vmbus/drivers/uio_hv_generic/new_id
```

If there already is a hardware specific kernel driver for the device, the generic driver still won't bind to it, in this case if you want to use the generic driver for a userspace library you'll have to manually unbind the hardware specific driver and bind the generic driver, using the device specific GUID like this:

```
echo -n ed963694-e847-4b2a-85af-bc9cfc11d6f3 > /sys/bus/vmbus/drivers/hv_netvsc/unbind
echo -n ed963694-e847-4b2a-85af-bc9cfc11d6f3 > /sys/bus/vmbus/drivers/uio_hv_generic/bind
```

You can verify that the device has been bound to the driver by looking for it in sysfs, for example like the following:

```
ls -l /sys/bus/vmbus/devices/ed963694-e847-4b2a-85af-bc9cfc11d6f3/driver
```

Which if successful should print:

```
.../ed963694-e847-4b2a-85af-bc9cfc11d6f3/driver -> ../../../bus/vmbus/drivers/uio_hv_generic
```

Things to know about uio_hv_generic

On each interrupt, uio_hv_generic sets the Interrupt Disable bit. This prevents the device from generating further interrupts until the bit is cleared. The userspace driver should clear this bit before blocking and waiting for more interrupts.

When host rescinds a device, the interrupt file descriptor is marked down and any reads of the interrupt file descriptor will return -EIO. Similar to a closed socket or disconnected serial device.

The vmbus device regions are mapped into uio device resources:

0. Channel ring buffers: guest to host and host to guest
1. Guest to host interrupt signalling pages
2. Guest to host monitor page
3. Network receive buffer region
4. Network send buffer region

Further information

- OSADL homepage.
- Linutronix homepage.

CHAPTER

THIRTYTWO

LINUX FIRMWARE API

Introduction

The firmware API enables kernel code to request files required for functionality from userspace, the uses vary:

- Microcode for CPU errata
- Device driver firmware, required to be loaded onto device microcontrollers
- Device driver information data (calibration data, EEPROM overrides), some of which can be completely optional.

Types of firmware requests

There are two types of calls:

- Synchronous
- Asynchronous

Which one you use vary depending on your requirements, the rule of thumb however is you should strive to use the asynchronous APIs unless you also are already using asynchronous initialization mechanisms which will not stall or delay boot. Even if loading firmware does not take a lot of time processing firmware might, and this can still delay boot or initialization, as such mechanisms such as asynchronous probe can help supplement drivers.

Firmware API core features

The firmware API has a rich set of core features available. This section documents these features.

Firmware search paths

The following search paths are used to look for firmware on your root filesystem.

- fw_path_para - module parameter - default is empty so this is ignored
- /lib/firmware/updates/UTS_RELEASE/
- /lib/firmware/updates/
- /lib/firmware/UTS_RELEASE/
- /lib/firmware/

The module parameter ''path" can be passed to the firmware_class module to activate the first optional custom fw_path_para. The custom path can only be up to 256 characters long. The kernel parameter passed would be:

- 'firmware_class.path=$CUSTOMIZED_PATH'

There is an alternative to customize the path at run time after bootup, you can use the file:

- /sys/module/firmware_class/parameters/path

You would echo into it your custom path and firmware requested will be searched for there first.

Built-in firmware

Firmware can be built-in to the kernel, this means building the firmware into vmlinux directly, to enable avoiding having to look for firmware from the filesystem. Instead, firmware can be looked for inside the kernel directly. You can enable built-in firmware using the kernel configuration options:

- CONFIG_EXTRA_FIRMWARE
- CONFIG_EXTRA_FIRMWARE_DIR

There are a few reasons why you might want to consider building your firmware into the kernel with CONFIG_EXTRA_FIRMWARE:

- Speed
- Firmware is needed for accessing the boot device, and the user doesn't want to stuff the firmware into the boot initramfs.

Even if you have these needs there are a few reasons why you may not be able to make use of built-in firmware:

- Legalese - firmware is non-GPL compatible
- Some firmware may be optional
- Firmware upgrades are possible, therefore a new firmware would implicate a complete kernel rebuild.
- Some firmware files may be really large in size. The remote-proc subsystem is an example subsystem which deals with these sorts of firmware
- The firmware may need to be scraped out from some device specific location dynamically, an example is calibration data for for some WiFi chipsets. This calibration data can be unique per sold device.

Firmware cache

When Linux resumes from suspend some device drivers require firmware lookups to re-initialize devices. During resume there may be a period of time during which firmware lookups are not possible, during this short period of time firmware requests will fail. Time is of essence though, and delaying drivers to wait for the root filesystem for firmware delays user experience with device functionality. In order to support these requirements the firmware infrastructure implements a firmware cache for device drivers for most API calls, automatically behind the scenes.

The firmware cache makes using certain firmware API calls safe during a device driver's suspend and resume callback. Users of these API calls needn't cache the firmware by themselves for dealing with firmware loss during system resume.

The firmware cache works by requesting for firmware prior to suspend and caching it in memory. Upon resume device drivers using the firmware API will have access to the firmware immediately, without having to wait for the root filesystem to mount or dealing with possible race issues with lookups as the root filesystem mounts.

Some implementation details about the firmware cache setup:

- The firmware cache is setup by adding a devres entry for each device that uses all synchronous call except `request_firmware_into_buf()`.

- If an asynchronous call is used the firmware cache is only set up for a device if if the second argument (uevent) to request_firmware_nowait() is true. When uevent is true it requests that a kobject uevent be sent to userspace for the firmware request. For details refer to the Fackback mechanism documented below.
- If the firmware cache is determined to be needed as per the above two criteria the firmware cache is setup by adding a devres entry for the device making the firmware request.
- The firmware devres entry is maintained throughout the lifetime of the device. This means that even if you release_firmware() the firmware cache will still be used on resume from suspend.
- The timeout for the fallback mechanism is temporarily reduced to 10 seconds as the firmware cache is set up during suspend, the timeout is set back to the old value you had configured after the cache is set up.
- Upon suspend any pending non-uevent firmware requests are killed to avoid stalling the kernel, this is done with kill_requests_without_uevent(). Kernel calls requiring the non-uevent therefore need to implement their own firmware cache mechanism but must not use the firmware API on suspend.

Direct filesystem lookup

Direct filesystem lookup is the most common form of firmware lookup performed by the kernel. The kernel looks for the firmware directly on the root filesystem in the paths documented in the section 'Firmware search paths'. The filesystem lookup is implemented in fw_get_filesystem_firmware(), it uses common core kernel file loader facility kernel_read_file_from_path(). The max path allowed is PATH_MAX – currently this is 4096 characters.

It is recommended you keep /lib/firmware paths on your root filesystem, avoid having a separate partition for them in order to avoid possible races with lookups and avoid uses of the custom fallback mechanisms documented below.

Firmware and initramfs

Drivers which are built-in to the kernel should have the firmware integrated also as part of the initramfs used to boot the kernel given that otherwise a race is possible with loading the driver and the real rootfs not yet being available. Stuffing the firmware into initramfs resolves this race issue, however note that using initrd does not suffice to address the same race.

There are circumstances that justify not wanting to include firmware into initramfs, such as dealing with large firmware firmware files for the remote-proc subsystem. For such cases using a userspace fallback mechanism is currently the only viable solution as only userspace can know for sure when the real rootfs is ready and mounted.

Fallback mechanisms

A fallback mechanism is supported to allow to overcome failures to do a direct filesystem lookup on the root filesystem or when the firmware simply cannot be installed for practical reasons on the root filesystem. The kernel configuration options related to supporting the firmware fallback mechanism are:

- CONFIG_FW_LOADER_USER_HELPER: enables building the firmware fallback mechanism. Most distributions enable this option today. If enabled but CONFIG_FW_LOADER_USER_HELPER_FALLBACK is disabled, only the custom fallback mechanism is available and for the request_firmware_nowait() call.
- CONFIG_FW_LOADER_USER_HELPER_FALLBACK: force enables each request to enable the kobject uevent fallback mechanism on all firmware API calls except request_firmware_direct(). Most distributions disable this option today. The call request_firmware_nowait() allows for one alternative fallback mechanism: if this kconfig option is enabled and your second argument to request_firmware_nowait(), uevent, is set to false you are informing the kernel that you have a custom fallback mechanism and it will manually load the firmware. Read below for more details.

Note that this means when having this configuration:

CONFIG_FW_LOADER_USER_HELPER=y CONFIG_FW_LOADER_USER_HELPER_FALLBACK=n

the kobject uevent fallback mechanism will never take effect even for request_firmware_nowait() when uevent is set to true.

Justifying the firmware fallback mechanism

Direct filesystem lookups may fail for a variety of reasons. Known reasons for this are worth itemizing and documenting as it justifies the need for the fallback mechanism:

- Race against access with the root filesystem upon bootup.
- Races upon resume from suspend. This is resolved by the firmware cache, but the firmware cache is only supported if you use uevents, and its not supported for request_firmware_into_buf().
- **Firmware is not accessible through typical means:**
    - It cannot be installed into the root filesystem
    - The firmware provides very unique device specific data tailored for the unit gathered with local information. An example is calibration data for WiFi chipsets for mobile devices. This calibration data is not common to all units, but tailored per unit. Such information may be installed on a separate flash partition other than where the root filesystem is provided.

Types of fallback mechanisms

There are really two fallback mechanisms available using one shared sysfs interface as a loading facility:

- Kobject uevent fallback mechanism
- Custom fallback mechanism

First lets document the shared sysfs loading facility.

Firmware sysfs loading facility

In order to help device drivers upload firmware using a fallback mechanism the firmware infrastructure creates a sysfs interface to enable userspace to load and indicate when firmware is ready. The sysfs directory is created via fw_create_instance(). This call creates a new struct device named after the firmware requested, and establishes it in the device hierarchy by associating the device used to make the request as the device's parent. The sysfs directory's file attributes are defined and controlled through the new device's class (firmware_class) and group (fw_dev_attr_groups). This is actually where the original firmware_class.c file name comes from, as originally the only firmware loading mechanism available was the mechanism we now use as a fallback mechanism.

To load firmware using the sysfs interface we expose a loading indicator, and a file upload firmware into:

- /sys/$DEVPATH/loading
- /sys/$DEVPATH/data

To upload firmware you will echo 1 onto the loading file to indicate you are loading firmware. You then cat the firmware into the data file, and you notify the kernel the firmware is ready by echo'ing 0 onto the loading file.

The firmware device used to help load firmware using sysfs is only created if direct firmware loading fails and if the fallback mechanism is enabled for your firmware request, this is set up with fw_load_from_user_helper(). It is important to re-iterate that no device is created if a direct filesystem lookup succeeded.

Using:

```
echo 1 > /sys/$DEVPATH/loading
```

Will clean any previous partial load at once and make the firmware API return an error. When loading firmware the firmware_class grows a buffer for the firmware in PAGE_SIZE increments to hold the image as it comes in.

firmware_data_read() and firmware_loading_show() are just provided for the test_firmware driver for testing, they are not called in normal use or expected to be used regularly by userspace.

Firmware kobject uevent fallback mechanism

Since a device is created for the sysfs interface to help load firmware as a fallback mechanism userspace can be informed of the addition of the device by relying on kobject uevents. The addition of the device into the device hierarchy means the fallback mechanism for firmware loading has been initiated. For details of implementation refer to _request_firmware_load(), in particular on the use of dev_set_uevent_suppress() and kobject_uevent().

The kernel's kobject uevent mechanism is implemented in lib/kobject_uevent.c, it issues uevents to userspace. As a supplement to kobject uevents Linux distributions could also enable CONFIG_UEVENT_HELPER_PATH, which makes use of core kernel's usermode helper (UMH) functionality to call out to a userspace helper for kobject uevents. In practice though no standard distribution has ever used the CONFIG_UEVENT_HELPER_PATH. If CONFIG_UEVENT_HELPER_PATH is enabled this binary would be called each time kobject_uevent_env() gets called in the kernel for each kobject uevent triggered.

Different implementations have been supported in userspace to take advantage of this fallback mechanism. When firmware loading was only possible using the sysfs mechanism the userspace component "hotplug" provided the functionality of monitoring for kobject events. Historically this was superseded be systemd's udev, however firmware loading support was removed from udev as of systemd commit be2ea723b1d0 ("udev: remove userspace firmware loading support") as of v217 on August, 2014. This means most Linux distributions today are not using or taking advantage of the firmware fallback mechanism provided by kobject uevents. This is specially exacerbated due to the fact that most distributions today disable CONFIG_FW_LOADER_USER_HELPER_FALLBACK.

Refer to do_firmware_uevent() for details of the kobject event variables setup. Variables passwdd with a kobject add event:

- FIRMWARE=firmware name
- TIMEOUT=timeout value
- ASYNC=whether or not the API request was asynchronous

By default DEVPATH is set by the internal kernel kobject infrastructure. Below is an example simple kobject uevent script:

```
# Both $DEVPATH and $FIRMWARE are already provided in the environment.
MY_FW_DIR=/lib/firmware/
echo 1 > /sys/$DEVPATH/loading
cat $MY_FW_DIR/$FIRMWARE > /sys/$DEVPATH/data
echo 0 > /sys/$DEVPATH/loading
```

Firmware custom fallback mechanism

Users of the request_firmware_nowait() call have yet another option available at their disposal: rely on the sysfs fallback mechanism but request that no kobject uevents be issued to userspace. The original logic behind this was that utilities other than udev might be required to lookup firmware in non-traditional paths – paths outside of the listing documented in the section 'Direct filesystem lookup'. This option is not available to any of the other API calls as uevents are always forced for them.

Since uevents are only meaningful if the fallback mechanism is enabled in your kernel it would seem odd to enable uevents with kernels that do not have the fallback mechanism enabled in their kernels. Unfortunately we also rely on the uevent flag which can be disabled by request_firmware_nowait() to also setup the firmware cache for firmware requests. As documented above, the firmware cache is only set up if uevent is enabled for an API call. Although this can disable the firmware cache for request_firmware_nowait() calls, users of this API should not use it for the purposes of disabling the cache as that was not the original purpose of the flag. Not setting the uevent flag means you want to opt-in for the firmware fallback mechanism but you want to suppress kobject uevents, as you have a custom solution which will monitor for your device addition into the device hierarchy somehow and load firmware for you through a custom path.

Firmware fallback timeout

The firmware fallback mechanism has a timeout. If firmware is not loaded onto the sysfs interface by the timeout value an error is sent to the driver. By default the timeout is set to 60 seconds if uevents are desirable, otherwise MAX_JIFFY_OFFSET is used (max timeout possible). The logic behind using MAX_JIFFY_OFFSET for non-uevents is that a custom solution will have as much time as it needs to load firmware.

You can customize the firmware timeout by echo'ing your desired timeout into the following file:

- /sys/class/firmware/timeout

If you echo 0 into it means MAX_JIFFY_OFFSET will be used. The data type for the timeout is an int.

Firmware lookup order

Different functionality is available to enable firmware to be found. Below is chronological order of how firmware will be looked for once a driver issues a firmware API call.

- The ''Built-in firmware" is checked first, if the firmware is present we return it immediately
- The ''Firmware cache" is looked at next. If the firmware is found we return it immediately
- The ''Direct filesystem lookup" is performed next, if found we return it immediately
- If no firmware has been found and the fallback mechanism was enabled the sysfs interface is created. After this either a kobject uevent is issued or the custom firmware loading is relied upon for firmware loading up to the timeout value.

request_firmware API

You would typically load firmware and then load it into your device somehow. The typical firmware work flow is reflected below:

```
if(request_firmware(&fw_entry, $FIRMWARE, device) == 0)
       copy_fw_to_device(fw_entry->data, fw_entry->size);
release_firmware(fw_entry);
```

Synchronous firmware requests

Synchronous firmware requests will wait until the firmware is found or until an error is returned.

request_firmware

int **request_firmware**(const struct firmware ** *firmware_p*, const char * *name*, struct *device* * *device*)
send firmware request and wait for it

Parameters

`const struct firmware ** firmware_p` pointer to firmware image

`const char * name` name of firmware file

`struct device * device` 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.

request_firmware_direct

int **request_firmware_direct**(const struct firmware ** *firmware_p*, const char * *name*, struct *device* * *device*)
load firmware directly without usermode helper

Parameters

`const struct firmware ** firmware_p` pointer to firmware image

`const char * name` name of firmware file

`struct device * device` 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.

request_firmware_into_buf

int **request_firmware_into_buf**(const struct firmware ** *firmware_p*, const char * *name*, struct *device* * *device*, void * *buf*, size_t *size*)
load firmware into a previously allocated buffer

Parameters

`const struct firmware ** firmware_p` pointer to firmware image

`const char * name` name of firmware file

`struct device * device` device for which firmware is being loaded and DMA region allocated

`void * buf` address of buffer to load firmware into

`size_t size` size of buffer

Description

This function works pretty much like *request_firmware()*, but it doesn't allocate a buffer to hold the firmware data. Instead, the firmware is loaded directly into the buffer pointed to by **buf** and the **firmware_p** data member is pointed at **buf**.

This function doesn't cache firmware either.

Asynchronous firmware requests

Asynchronous firmware requests allow driver code to not have to wait until the firmware or an error is returned. Function callbacks are provided so that when the firmware or an error is found the driver is informed through the callback. request_firmware_nowait() cannot be called in atomic contexts.

request_firmware_nowait

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*))

asynchronous version of request_firmware

Parameters

struct module * module module requesting the firmware

bool uevent sends uevent to copy the firmware image if this flag is non-zero else the firmware copy must be done manually.

const char * name name of firmware file

struct device * device device for which firmware is being loaded

gfp_t gfp allocation flags

void * context will be passed over to **cont**, and **fw** may be NULL if firmware request fails.

void (*)(const struct firmware *fw, void *context) cont 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 ->:c:func:*probe()* methods, if **gfp** is GFP_KERNEL.
- can't sleep at all if **gfp** is GFP_ATOMIC.

request firmware API expected driver use

Once an API call returns you process the firmware and then release the firmware. For example if you used request_firmware() and it returns, the driver has the firmware image accessible in fw_entry->{data,size}. If something went wrong request_firmware() returns non-zero and fw_entry is set to NULL. Once your driver is done with processing the firmware it can call call release_firmware(fw_entry) to release the firmware image and any related resource.

Other Firmware Interfaces

DMI Interfaces

int **dmi_check_system**(const struct dmi_system_id * *list*)
check system DMI data

Parameters

const struct dmi_system_id * list array of dmi_system_id structures to match against All non-null elements of the list must match their slot's (field index's) data (i.e., each list string must be a substring of the specified DMI slot's string data) to be considered a successful match.

Description

Walk the blacklist table running matching functions until someone returns non zero or we hit the end. Callback function is called for each successful match. Returns the number of matches.

dmi_scan_machine must be called before this function is called.

const struct dmi_system_id * **dmi_first_match**(const struct dmi_system_id * *list*)
find dmi_system_id structure matching system DMI data

Parameters

const struct dmi_system_id * list array of dmi_system_id structures to match against All non-null elements of the list must match their slot's (field index's) data (i.e., each list string must be a substring of the specified DMI slot's string data) to be considered a successful match.

Description

Walk the blacklist table until the first match is found. Return the pointer to the matching entry or NULL if there's no match.

dmi_scan_machine must be called before this function is called.

const char * **dmi_get_system_info**(int *field*)
return DMI data value

Parameters

int field data index (see enum dmi_field)

Description

Returns one DMI data value, can be used to perform complex DMI data checks.

int **dmi_name_in_vendors**(const char * *str*)
Check if string is in the DMI system or board vendor name

Parameters

const char * str Case sensitive Name

const struct dmi_device * **dmi_find_device**(int *type*, const char * *name*, const struct dmi_device * *from*)
find onboard device by type/name

Parameters

int type device type or `DMI_DEV_TYPE_ANY` to match all device types

const char * name device name string or `NULL` to match all

const struct dmi_device * from previous device found in search, or `NULL` for new search.

Description

Iterates through the list of known onboard devices. If a device is found with a matching **type** and **name**, a pointer to its device structure is returned. Otherwise, NULL is returned. A new search is initiated by passing NULL as the **from** argument. If **from** is not NULL, searches continue from next device.

bool **dmi_get_date**(int *field*, int * *yearp*, int * *monthp*, int * *dayp*)
parse a DMI date

Parameters

int field data index (see enum dmi_field)

int * yearp optional out parameter for the year

int * monthp optional out parameter for the month

int * dayp optional out parameter for the day

Description

The date field is assumed to be in the form resembling [mm[/dd]]/yy[yy] and the result is stored in the out parameters any or all of which can be omitted.

If the field doesn't exist, all out parameters are set to zero and false is returned. Otherwise, true is returned with any invalid part of date set to zero.

On return, year, month and day are guaranteed to be in the range of [0,9999], [0,12] and [0,31] respectively.

int **dmi_walk**(void (*decode) (const struct dmi_header *, void *, void * *private_data*)
Walk the DMI table and get called back for every record

Parameters

void (*)(const struct dmi_header *, void *) decode Callback function

void * private_data Private data to be passed to the callback function

Description

Returns 0 on success, -ENXIO if DMI is not selected or not present, or a different negative error code if DMI walking fails.

bool **dmi_match**(enum dmi_field *f*, const char * *str*)
compare a string to the dmi field (if exists)

Parameters

enum dmi_field f DMI field identifier

const char * str string to compare the DMI field to

Description

Returns true if the requested field equals to the str (including NULL).

EDD Interfaces

ssize_t **edd_show_raw_data**(struct edd_device * *edev*, char * *buf*)
copies raw data to buffer for userspace to parse

Parameters

struct edd_device * edev target edd_device

char * buf output buffer

Return

number of bytes written, or -EINVAL on failure

void **edd_release**(struct kobject * *kobj*)
free edd structure

Parameters

struct kobject * kobj kobject of edd structure

Description

This is called when the refcount of the edd structure reaches 0. This should happen right after we unregister, but just in case, we use the release callback anyway.

int **edd_dev_is_type**(struct edd_device * *edev*, const char * *type*)
is this EDD device a 'type' device?

Parameters

struct edd_device * edev target edd_device

const char * type a host bus or interface identifier string per the EDD spec

Description

Returns 1 (TRUE) if it is a 'type' device, 0 otherwise.

struct pci_dev * **edd_get_pci_dev**(struct edd_device * *edev*)
finds pci_dev that matches edev

Parameters

struct edd_device * edev edd_device

Description

Returns pci_dev if found, or NULL

int **edd_init**(void)
creates sysfs tree of EDD data

Parameters

void no arguments

CHAPTER

THIRTYTHREE

PINCTRL (PIN CONTROL) SUBSYSTEM

This document outlines the pin control subsystem in Linux

This subsystem deals with:

- Enumerating and naming controllable pins
- Multiplexing of pins, pads, fingers (etc) see below for details
- Configuration of pins, pads, fingers (etc), such as software-controlled biasing and driving mode specific pins, such as pull-up/down, open drain, load capacitance etc.

Top-level interface

Definition of PIN CONTROLLER:

- A pin controller is a piece of hardware, usually a set of registers, that can control PINs. It may be able to multiplex, bias, set load capacitance, set drive strength, etc. for individual pins or groups of pins.

Definition of PIN:

- PINS are equal to pads, fingers, balls or whatever packaging input or output line you want to control and these are denoted by unsigned integers in the range 0..maxpin. This numberspace is local to each PIN CONTROLLER, so there may be several such number spaces in a system. This pin space may be sparse - i.e. there may be gaps in the space with numbers where no pin exists.

When a PIN CONTROLLER is instantiated, it will register a descriptor to the pin control framework, and this descriptor contains an array of pin descriptors describing the pins handled by this specific pin controller.

Here is an example of a PGA (Pin Grid Array) chip seen from underneath:

```
     A   B   C   D   E   F   G   H

8    o   o   o   o   o   o   o   o

7    o   o   o   o   o   o   o   o

6    o   o   o   o   o   o   o   o

5    o   o   o   o   o   o   o   o

4    o   o   o   o   o   o   o   o

3    o   o   o   o   o   o   o   o

2    o   o   o   o   o   o   o   o

1    o   o   o   o   o   o   o   o
```

To register a pin controller and name all the pins on this package we can do this in our driver:

```
#include <linux/pinctrl/pinctrl.h>

const struct pinctrl_pin_desc foo_pins[] = {
        PINCTRL_PIN(0, "A8"),
        PINCTRL_PIN(1, "B8"),
        PINCTRL_PIN(2, "C8"),
        ...
        PINCTRL_PIN(61, "F1"),
        PINCTRL_PIN(62, "G1"),
        PINCTRL_PIN(63, "H1"),
};

static struct pinctrl_desc foo_desc = {
        .name = "foo",
        .pins = foo_pins,
        .npins = ARRAY_SIZE(foo_pins),
        .owner = THIS_MODULE,
};

int __init foo_probe(void)
{
        int error;

        struct pinctrl_dev *pctl;

        error = pinctrl_register_and_init(&foo_desc, <PARENT>,
                                          NULL, &pctl);
        if (error)
                return error;

        return pinctrl_enable(pctl);
}
```

To enable the pinctrl subsystem and the subgroups for PINMUX and PINCONF and selected drivers, you need to select them from your machine's Kconfig entry, since these are so tightly integrated with the machines they are used on. See for example arch/arm/mach-u300/Kconfig for an example.

Pins usually have fancier names than this. You can find these in the datasheet for your chip. Notice that the core pinctrl.h file provides a fancy macro called PINCTRL_PIN() to create the struct entries. As you can see I enumerated the pins from 0 in the upper left corner to 63 in the lower right corner. This enumeration was arbitrarily chosen, in practice you need to think through your numbering system so that it matches the layout of registers and such things in your driver, or the code may become complicated. You must also consider matching of offsets to the GPIO ranges that may be handled by the pin controller.

For a padring with 467 pads, as opposed to actual pins, I used an enumeration like this, walking around the edge of the chip, which seems to be industry standard too (all these pads had names, too):

```
  0 ..... 104
466        105
  .        .
  .        .
358        224
 357 .... 225
```

Pin groups

Many controllers need to deal with groups of pins, so the pin controller subsystem has a mechanism for enumerating groups of pins and retrieving the actual enumerated pins that are part of a certain group.

For example, say that we have a group of pins dealing with an SPI interface on { 0, 8, 16, 24 }, and a group of pins dealing with an I2C interface on pins on { 24, 25 }.

These two groups are presented to the pin control subsystem by implementing some generic pinctrl_ops like this:

```
#include <linux/pinctrl/pinctrl.h>

struct foo_group {
        const char *name;
        const unsigned int *pins;
        const unsigned num_pins;
};

static const unsigned int spi0_pins[] = { 0, 8, 16, 24 };
static const unsigned int i2c0_pins[] = { 24, 25 };

static const struct foo_group foo_groups[] = {
        {
                .name = "spi0_grp",
                .pins = spi0_pins,
                .num_pins = ARRAY_SIZE(spi0_pins),
        },
        {
                .name = "i2c0_grp",
                .pins = i2c0_pins,
                .num_pins = ARRAY_SIZE(i2c0_pins),
        },
};


static int foo_get_groups_count(struct pinctrl_dev *pctldev)
{
        return ARRAY_SIZE(foo_groups);
}

static const char *foo_get_group_name(struct pinctrl_dev *pctldev,
                                unsigned selector)
{
        return foo_groups[selector].name;
}

static int foo_get_group_pins(struct pinctrl_dev *pctldev, unsigned selector,
                        const unsigned **pins,
                        unsigned *num_pins)
{
        *pins = (unsigned *) foo_groups[selector].pins;
        *num_pins = foo_groups[selector].num_pins;
        return 0;
}

static struct pinctrl_ops foo_pctrl_ops = {
        .get_groups_count = foo_get_groups_count,
        .get_group_name = foo_get_group_name,
        .get_group_pins = foo_get_group_pins,
};


static struct pinctrl_desc foo_desc = {
...
.pctlops = &foo_pctrl_ops,
};
```

The pin control subsystem will call the .get_groups_count() function to determine the total number of legal selectors, then it will call the other functions to retrieve the name and pins of the group. Maintaining the data structure of the groups is up to the driver, this is just a simple example - in practice you may need more entries in your group structure, for example specific register ranges associated with each group and so on.

Pin configuration

Pins can sometimes be software-configured in various ways, mostly related to their electronic properties when used as inputs or outputs. For example you may be able to make an output pin high impedance, or "tristate" meaning it is effectively disconnected. You may be able to connect an input pin to VDD or GND using a certain resistor value - pull up and pull down - so that the pin has a stable value when nothing is driving the rail it is connected to, or when it's unconnected.

Pin configuration can be programmed by adding configuration entries into the mapping table; see section "Board/machine configuration" below.

The format and meaning of the configuration parameter, PLATFORM_X_PULL_UP above, is entirely defined by the pin controller driver.

The pin configuration driver implements callbacks for changing pin configuration in the pin controller ops like this:

```
#include <linux/pinctrl/pinctrl.h>
#include <linux/pinctrl/pinconf.h>
#include "platform_x_pindefs.h"

static int foo_pin_config_get(struct pinctrl_dev *pctldev,
                unsigned offset,
                unsigned long *config)
{
        struct my_conftype conf;

        ... Find setting for pin @ offset ...

        *config = (unsigned long) conf;
}

static int foo_pin_config_set(struct pinctrl_dev *pctldev,
                unsigned offset,
                unsigned long config)
{
        struct my_conftype *conf = (struct my_conftype *) config;

        switch (conf) {
                case PLATFORM_X_PULL_UP:
                ...
                }
        }
}

static int foo_pin_config_group_get (struct pinctrl_dev *pctldev,
                unsigned selector,
                unsigned long *config)
{
        ...
}

static int foo_pin_config_group_set (struct pinctrl_dev *pctldev,
                unsigned selector,
                unsigned long config)
{
        ...
}
```

```
static struct pinconf_ops foo_pconf_ops = {
        .pin_config_get = foo_pin_config_get,
        .pin_config_set = foo_pin_config_set,
        .pin_config_group_get = foo_pin_config_group_get,
        .pin_config_group_set = foo_pin_config_group_set,
};

/* Pin config operations are handled by some pin controller */
static struct pinctrl_desc foo_desc = {
        ...
        .confops = &foo_pconf_ops,
};
```

Since some controllers have special logic for handling entire groups of pins they can exploit the special whole-group pin control function. The pin_config_group_set() callback is allowed to return the error code -EAGAIN, for groups it does not want to handle, or if it just wants to do some group-level handling and then fall through to iterate over all pins, in which case each individual pin will be treated by separate pin_config_set() calls as well.

Interaction with the GPIO subsystem

The GPIO drivers may want to perform operations of various types on the same physical pins that are also registered as pin controller pins.

First and foremost, the two subsystems can be used as completely orthogonal, see the section named “pin control requests from drivers” and “drivers needing both pin control and GPIOs” below for details. But in some situations a cross-subsystem mapping between pins and GPIOs is needed.

Since the pin controller subsystem has its pinspace local to the pin controller we need a mapping so that the pin control subsystem can figure out which pin controller handles control of a certain GPIO pin. Since a single pin controller may be muxing several GPIO ranges (typically SoCs that have one set of pins, but internally several GPIO silicon blocks, each modelled as a struct gpio_chip) any number of GPIO ranges can be added to a pin controller instance like this:

```
struct gpio_chip chip_a;
struct gpio_chip chip_b;

static struct pinctrl_gpio_range gpio_range_a = {
        .name = "chip a",
        .id = 0,
        .base = 32,
        .pin_base = 32,
        .npins = 16,
        .gc = &chip_a;
};

static struct pinctrl_gpio_range gpio_range_b = {
        .name = "chip b",
        .id = 0,
        .base = 48,
        .pin_base = 64,
        .npins = 8,
        .gc = &chip_b;
};

{
        struct pinctrl_dev *pctl;
        ...
        pinctrl_add_gpio_range(pctl, &gpio_range_a);
        pinctrl_add_gpio_range(pctl, &gpio_range_b);
}
```

So this complex system has one pin controller handling two different GPIO chips. “chip a” has 16 pins and “chip b” has 8 pins. The “chip a” and “chip b” have different .pin_base, which means a start pin number of the GPIO range.

The GPIO range of “chip a” starts from the GPIO base of 32 and actual pin range also starts from 32. However “chip b” has different starting offset for the GPIO range and pin range. The GPIO range of “chip b” starts from GPIO number 48, while the pin range of “chip b” starts from 64.

We can convert a gpio number to actual pin number using this “pin_base”. They are mapped in the global GPIO pin space at:

chip a:

- GPIO range : [32 .. 47]
- pin range : [32 .. 47]

chip b:

- GPIO range : [48 .. 55]
- pin range : [64 .. 71]

The above examples assume the mapping between the GPIOs and pins is linear. If the mapping is sparse or haphazard, an array of arbitrary pin numbers can be encoded in the range like this:

```
static const unsigned range_pins[] = { 14, 1, 22, 17, 10, 8, 6, 2 };

static struct pinctrl_gpio_range gpio_range = {
        .name = "chip",
        .id = 0,
        .base = 32,
        .pins = &range_pins,
        .npins = ARRAY_SIZE(range_pins),
        .gc = &chip;
};
```

In this case the pin_base property will be ignored. If the name of a pin group is known, the pins and npins elements of the above structure can be initialised using the function pinctrl_get_group_pins(), e.g. for pin group “foo”:

```
pinctrl_get_group_pins(pctl, "foo", &gpio_range.pins,
                       &gpio_range.npins);
```

When GPIO-specific functions in the pin control subsystem are called, these ranges will be used to look up the appropriate pin controller by inspecting and matching the pin to the pin ranges across all controllers. When a pin controller handling the matching range is found, GPIO-specific functions will be called on that specific pin controller.

For all functionalities dealing with pin biasing, pin muxing etc, the pin controller subsystem will look up the corresponding pin number from the passed in gpio number, and use the range’s internals to retrieve a pin number. After that, the subsystem passes it on to the pin control driver, so the driver will get a pin number into its handled number range. Further it is also passed the range ID value, so that the pin controller knows which range it should deal with.

Calling pinctrl_add_gpio_range from pinctrl driver is DEPRECATED. Please see section 2.1 of Documentation/devicetree/bindings/gpio/gpio.txt on how to bind pinctrl and gpio drivers.

PINMUX interfaces

These calls use the pinmux_* naming prefix. No other calls should use that prefix.

What is pinmuxing?

PINMUX, also known as padmux, ballmux, alternate functions or mission modes is a way for chip vendors producing some kind of electrical packages to use a certain physical pin (ball, pad, finger, etc) for multiple mutually exclusive functions, depending on the application. By "application" in this context we usually mean a way of soldering or wiring the package into an electronic system, even though the framework makes it possible to also change the function at runtime.

Here is an example of a PGA (Pin Grid Array) chip seen from underneath:

```
     A   B   C   D   E   F   G   H
   +---+
8  | o | o   o   o   o   o   o   o
   |   |
7  | o | o   o   o   o   o   o   o
   |   |
6  | o | o   o   o   o   o   o   o
   +---+---+
5  | o | o | o   o   o   o   o   o
   +---+---+               +---+
4    o   o   o   o   o   o | o | o
                           |   |
3    o   o   o   o   o   o | o | o
                           |   |
2    o   o   o   o   o   o | o | o
   +-------+-------+-------+---+---+
1  | o   o | o   o | o   o | o | o |
   +-------+-------+-------+---+---+
```

This is not tetris. The game to think of is chess. Not all PGA/BGA packages are chessboard-like, big ones have "holes" in some arrangement according to different design patterns, but we're using this as a simple example. Of the pins you see some will be taken by things like a few VCC and GND to feed power to the chip, and quite a few will be taken by large ports like an external memory interface. The remaining pins will often be subject to pin multiplexing.

The example 8x8 PGA package above will have pin numbers 0 through 63 assigned to its physical pins. It will name the pins { A1, A2, A3 ... H6, H7, H8 } using pinctrl_register_pins() and a suitable data set as shown earlier.

In this 8x8 BGA package the pins { A8, A7, A6, A5 } can be used as an SPI port (these are four pins: CLK, RXD, TXD, FRM). In that case, pin B5 can be used as some general-purpose GPIO pin. However, in another setting, pins { A5, B5 } can be used as an I2C port (these are just two pins: SCL, SDA). Needless to say, we cannot use the SPI port and I2C port at the same time. However in the inside of the package the silicon performing the SPI logic can alternatively be routed out on pins { G4, G3, G2, G1 }.

On the bottom row at { A1, B1, C1, D1, E1, F1, G1, H1 } we have something special - it's an external MMC bus that can be 2, 4 or 8 bits wide, and it will consume 2, 4 or 8 pins respectively, so either { A1, B1 } are taken or { A1, B1, C1, D1 } or all of them. If we use all 8 bits, we cannot use the SPI port on pins { G4, G3, G2, G1 } of course.

This way the silicon blocks present inside the chip can be multiplexed "muxed" out on different pin ranges. Often contemporary SoC (systems on chip) will contain several I2C, SPI, SDIO/MMC, etc silicon blocks that can be routed to different pins by pinmux settings.

Since general-purpose I/O pins (GPIO) are typically always in shortage, it is common to be able to use almost any pin as a GPIO pin if it is not currently in use by some other I/O port.

Pinmux conventions

The purpose of the pinmux functionality in the pin controller subsystem is to abstract and provide pinmux settings to the devices you choose to instantiate in your machine configuration. It is inspired by the clk, GPIO and regulator subsystems, so devices will request their mux setting, but it's also possible to request a single pin for e.g. GPIO.

Definitions:

- FUNCTIONS can be switched in and out by a driver residing with the pin control subsystem in the drivers/pinctrl/* directory of the kernel. The pin control driver knows the possible functions. In the example above you can identify three pinmux functions, one for spi, one for i2c and one for mmc.
- FUNCTIONS are assumed to be enumerable from zero in a one-dimensional array. In this case the array could be something like: { spi0, i2c0, mmc0 } for the three available functions.
- FUNCTIONS have PIN GROUPS as defined on the generic level - so a certain function is *always* associated with a certain set of pin groups, could be just a single one, but could also be many. In the example above the function i2c is associated with the pins { A5, B5 }, enumerated as { 24, 25 } in the controller pin space.

  The Function spi is associated with pin groups { A8, A7, A6, A5 } and { G4, G3, G2, G1 }, which are enumerated as { 0, 8, 16, 24 } and { 38, 46, 54, 62 } respectively.

  Group names must be unique per pin controller, no two groups on the same controller may have the same name.
- The combination of a FUNCTION and a PIN GROUP determine a certain function for a certain set of pins. The knowledge of the functions and pin groups and their machine-specific particulars are kept inside the pinmux driver, from the outside only the enumerators are known, and the driver core can request:
  - The name of a function with a certain selector (>= 0)
  - A list of groups associated with a certain function
  - That a certain group in that list to be activated for a certain function

  As already described above, pin groups are in turn self-descriptive, so the core will retrieve the actual pin range in a certain group from the driver.
- FUNCTIONS and GROUPS on a certain PIN CONTROLLER are MAPPED to a certain device by the board file, device tree or similar machine setup configuration mechanism, similar to how regulators are connected to devices, usually by name. Defining a pin controller, function and group thus uniquely identify the set of pins to be used by a certain device. (If only one possible group of pins is available for the function, no group name need to be supplied - the core will simply select the first and only group available.)

  In the example case we can define that this particular machine shall use device spi0 with pinmux function fspi0 group gspi0 and i2c0 on function fi2c0 group gi2c0, on the primary pin controller, we get mappings like these:

```
{
        {"map-spi0", spi0, pinctrl0, fspi0, gspi0},
        {"map-i2c0", i2c0, pinctrl0, fi2c0, gi2c0}
}
```

  Every map must be assigned a state name, pin controller, device and function. The group is not compulsory - if it is omitted the first group presented by the driver as applicable for the function will be selected, which is useful for simple cases.

  It is possible to map several groups to the same combination of device, pin controller and function. This is for cases where a certain function on a certain pin controller may use different sets of pins in different configurations.
- PINS for a certain FUNCTION using a certain PIN GROUP on a certain PIN CONTROLLER are provided on a first-come first-serve basis, so if some other device mux setting or GPIO pin request has already taken your physical pin, you will be denied the use of it. To get (activate) a new setting, the old one has to be put (deactivated) first.

Sometimes the documentation and hardware registers will be oriented around pads (or "fingers") rather than pins - these are the soldering surfaces on the silicon inside the package, and may or may not match the actual number of pins/balls underneath the capsule. Pick some enumeration that makes sense to you. Define enumerators only for the pins you can control if that makes sense.

Assumptions:

We assume that the number of possible function maps to pin groups is limited by the hardware. I.e. we assume that there is no system where any function can be mapped to any pin, like in a phone exchange. So the available pin groups for a certain function will be limited to a few choices (say up to eight or so), not hundreds or any amount of choices. This is the characteristic we have found by inspecting available pinmux hardware, and a necessary assumption since we expect pinmux drivers to present *all* possible function vs pin group mappings to the subsystem.

Pinmux drivers

The pinmux core takes care of preventing conflicts on pins and calling the pin controller driver to execute different settings.

It is the responsibility of the pinmux driver to impose further restrictions (say for example infer electronic limitations due to load, etc.) to determine whether or not the requested function can actually be allowed, and in case it is possible to perform the requested mux setting, poke the hardware so that this happens.

Pinmux drivers are required to supply a few callback functions, some are optional. Usually the set_mux() function is implemented, writing values into some certain registers to activate a certain mux setting for a certain pin.

A simple driver for the above example will work by setting bits 0, 1, 2, 3 or 4 into some register named MUX to select a certain function with a certain group of pins would work something like this:

```
#include <linux/pinctrl/pinctrl.h>
#include <linux/pinctrl/pinmux.h>

struct foo_group {
        const char *name;
        const unsigned int *pins;
        const unsigned num_pins;
};

static const unsigned spi0_0_pins[] = { 0, 8, 16, 24 };
static const unsigned spi0_1_pins[] = { 38, 46, 54, 62 };
static const unsigned i2c0_pins[] = { 24, 25 };
static const unsigned mmc0_1_pins[] = { 56, 57 };
static const unsigned mmc0_2_pins[] = { 58, 59 };
static const unsigned mmc0_3_pins[] = { 60, 61, 62, 63 };

static const struct foo_group foo_groups[] = {
        {
                .name = "spi0_0_grp",
                .pins = spi0_0_pins,
                .num_pins = ARRAY_SIZE(spi0_0_pins),
        },
        {
                .name = "spi0_1_grp",
                .pins = spi0_1_pins,
                .num_pins = ARRAY_SIZE(spi0_1_pins),
        },
        {
                .name = "i2c0_grp",
                .pins = i2c0_pins,
                .num_pins = ARRAY_SIZE(i2c0_pins),
        },
        {
                .name = "mmc0_1_grp",
                .pins = mmc0_1_pins,
                .num_pins = ARRAY_SIZE(mmc0_1_pins),
        },
        {
                .name = "mmc0_2_grp",
                .pins = mmc0_2_pins,
                .num_pins = ARRAY_SIZE(mmc0_2_pins),
        },
        {
                .name = "mmc0_3_grp",
                .pins = mmc0_3_pins,
                .num_pins = ARRAY_SIZE(mmc0_3_pins),
        },
};


static int foo_get_groups_count(struct pinctrl_dev *pctldev)
{
        return ARRAY_SIZE(foo_groups);
}

static const char *foo_get_group_name(struct pinctrl_dev *pctldev,
                                unsigned selector)
{
        return foo_groups[selector].name;
}

static int foo_get_group_pins(struct pinctrl_dev *pctldev, unsigned selector,
                        unsigned ** const pins,
                        unsigned * const num_pins)
{
        *pins = (unsigned *) foo_groups[selector].pins;
        *num_pins = foo_groups[selector].num_pins;
        return 0;
}

static struct pinctrl_ops foo_pctrl_ops = {
        .get_groups_count = foo_get_groups_count,
        .get_group_name = foo_get_group_name,
        .get_group_pins = foo_get_group_pins,
};

struct foo_pmx_func {
        const char *name;
        const char * const *groups;
        const unsigned num_groups;
};

static const char * const spi0_groups[] = { "spi0_0_grp", "spi0_1_grp" };
static const char * const i2c0_groups[] = { "i2c0_grp" };
static const char * const mmc0_groups[] = { "mmc0_1_grp", "mmc0_2_grp",
                                        "mmc0_3_grp" };

static const struct foo_pmx_func foo_functions[] = {
        {
                .name = "spi0",
                .groups = spi0_groups,
                .num_groups = ARRAY_SIZE(spi0_groups),
        },
        {
                .name = "i2c0",
		.groups = i2c0_groups,
		.num_groups = ARRAY_SIZE(i2c0_groups),
	},
	{
		.name = "mmc0",
		.groups = mmc0_groups,
		.num_groups = ARRAY_SIZE(mmc0_groups),
	},
};

static int foo_get_functions_count(struct pinctrl_dev *pctldev)
{
	return ARRAY_SIZE(foo_functions);
}

static const char *foo_get_fname(struct pinctrl_dev *pctldev, unsigned selector)
{
	return foo_functions[selector].name;
}

static int foo_get_groups(struct pinctrl_dev *pctldev, unsigned selector,
			  const char * const **groups,
			  unsigned * const num_groups)
{
	*groups = foo_functions[selector].groups;
	*num_groups = foo_functions[selector].num_groups;
	return 0;
}

static int foo_set_mux(struct pinctrl_dev *pctldev, unsigned selector,
		unsigned group)
{
	u8 regbit = (1 << selector + group);

	writeb((readb(MUX)|regbit), MUX)
	return 0;
}

static struct pinmux_ops foo_pmxops = {
	.get_functions_count = foo_get_functions_count,
	.get_function_name = foo_get_fname,
	.get_function_groups = foo_get_groups,
	.set_mux = foo_set_mux,
	.strict = true,
};

/* Pinmux operations are handled by some pin controller */
static struct pinctrl_desc foo_desc = {
	...
	.pctlops = &foo_pctrl_ops,
	.pmxops = &foo_pmxops,
};
```

In the example activating muxing 0 and 1 at the same time setting bits 0 and 1, uses one pin in common so they would collide.

The beauty of the pinmux subsystem is that since it keeps track of all pins and who is using them, it will already have denied an impossible request like that, so the driver does not need to worry about such things - when it gets a selector passed in, the pinmux subsystem makes sure no other device or GPIO assignment is already using the selected pins. Thus bits 0 and 1 in the control register will never be set at the same time.

All the above functions are mandatory to implement for a pinmux driver.

Pin control interaction with the GPIO subsystem

Note that the following implies that the use case is to use a certain pin from the Linux kernel using the API in <linux/gpio.h> with gpio_request() and similar functions. There are cases where you may be using something that your datasheet calls "GPIO mode", but actually is just an electrical configuration for a certain device. See the section below named "GPIO mode pitfalls" for more details on this scenario.

The public pinmux API contains two functions named pinctrl_gpio_request() and pinctrl_gpio_free(). These two functions shall *ONLY* be called from gpiolib-based drivers as part of their gpio_request() and gpio_free() semantics. Likewise the pinctrl_gpio_direction_[input|output] shall only be called from within respective gpio_direction_[input|output] gpiolib implementation.

NOTE that platforms and individual drivers shall *NOT* request GPIO pins to be controlled e.g. muxed in. Instead, implement a proper gpiolib driver and have that driver request proper muxing and other control for its pins.

The function list could become long, especially if you can convert every individual pin into a GPIO pin independent of any other pins, and then try the approach to define every pin as a function.

In this case, the function array would become 64 entries for each GPIO setting and then the device functions.

For this reason there are two functions a pin control driver can implement to enable only GPIO on an individual pin: .gpio_request_enable() and .gpio_disable_free().

This function will pass in the affected GPIO range identified by the pin controller core, so you know which GPIO pins are being affected by the request operation.

If your driver needs to have an indication from the framework of whether the GPIO pin shall be used for input or output you can implement the .gpio_set_direction() function. As described this shall be called from the gpiolib driver and the affected GPIO range, pin offset and desired direction will be passed along to this function.

Alternatively to using these special functions, it is fully allowed to use named functions for each GPIO pin, the pinctrl_gpio_request() will attempt to obtain the function "gpioN" where "N" is the global GPIO pin number if no special GPIO-handler is registered.

GPIO mode pitfalls

Due to the naming conventions used by hardware engineers, where "GPIO" is taken to mean different things than what the kernel does, the developer may be confused by a datasheet talking about a pin being possible to set into "GPIO mode". It appears that what hardware engineers mean with "GPIO mode" is not necessarily the use case that is implied in the kernel interface <linux/gpio.h>: a pin that you grab from kernel code and then either listen for input or drive high/low to assert/deassert some external line.

Rather hardware engineers think that "GPIO mode" means that you can software-control a few electrical properties of the pin that you would not be able to control if the pin was in some other mode, such as muxed in for a device.

The GPIO portions of a pin and its relation to a certain pin controller configuration and muxing logic can be constructed in several ways. Here are two examples:

```
(A)
                  pin config
                  logic regs
                  |               +- SPI
Physical pins --- pad --- pinmux -+- I2C
                          |       +- mmc
                          |       +- GPIO
                          pin
                          multiplex
                          logic regs
```

Here some electrical properties of the pin can be configured no matter whether the pin is used for GPIO or not. If you multiplex a GPIO onto a pin, you can also drive it high/low from "GPIO" registers. Alternatively, the pin can be controlled by a certain peripheral, while still applying desired pin config properties. GPIO functionality is thus orthogonal to any other device using the pin.

In this arrangement the registers for the GPIO portions of the pin controller, or the registers for the GPIO hardware module are likely to reside in a separate memory range only intended for GPIO driving, and the register range dealing with pin config and pin multiplexing get placed into a different memory range and a separate section of the data sheet.

A flag "strict" in struct pinmux_ops is available to check and deny simultaneous access to the same pin from GPIO and pin multiplexing consumers on hardware of this type. The pinctrl driver should set this flag accordingly.

```
(B)

                  pin config
                  logic regs
                  |               +- SPI
Physical pins --- pad --- pinmux -+- I2C
                  |       |       +- mmc
                  |       |
                  GPIO    pin
                          multiplex
                          logic regs
```

In this arrangement, the GPIO functionality can always be enabled, such that e.g. a GPIO input can be used to "spy" on the SPI/I2C/MMC signal while it is pulsed out. It is likely possible to disrupt the traffic on the pin by doing wrong things on the GPIO block, as it is never really disconnected. It is possible that the GPIO, pin config and pin multiplex registers are placed into the same memory range and the same section of the data sheet, although that need not be the case.

In some pin controllers, although the physical pins are designed in the same way as (B), the GPIO function still can't be enabled at the same time as the peripheral functions. So again the "strict" flag should be set, denying simultaneous activation by GPIO and other muxed in devices.

From a kernel point of view, however, these are different aspects of the hardware and shall be put into different subsystems:

- Registers (or fields within registers) that control electrical properties of the pin such as biasing and drive strength should be exposed through the pinctrl subsystem, as "pin configuration" settings.
- Registers (or fields within registers) that control muxing of signals from various other HW blocks (e.g. I2C, MMC, or GPIO) onto pins should be exposed through the pinctrl subsystem, as mux functions.
- Registers (or fields within registers) that control GPIO functionality such as setting a GPIO's output value, reading a GPIO's input value, or setting GPIO pin direction should be exposed through the GPIO subsystem, and if they also support interrupt capabilities, through the irqchip abstraction.

Depending on the exact HW register design, some functions exposed by the GPIO subsystem may call into the pinctrl subsystem in order to co-ordinate register settings across HW modules. In particular, this may be needed for HW with separate GPIO and pin controller HW modules, where e.g. GPIO direction is determined by a register in the pin controller HW module rather than the GPIO HW module.

Electrical properties of the pin such as biasing and drive strength may be placed at some pin-specific register in all cases or as part of the GPIO register in case (B) especially. This doesn't mean that such properties necessarily pertain to what the Linux kernel calls "GPIO".

Example: a pin is usually muxed in to be used as a UART TX line. But during system sleep, we need to put this pin into "GPIO mode" and ground it.

If you make a 1-to-1 map to the GPIO subsystem for this pin, you may start to think that you need to come up with something really complex, that the pin shall be used for UART TX and GPIO at the same time, that you will grab a pin control handle and set it to a certain state to enable UART TX to be muxed in, then twist it over to GPIO mode and use gpio_direction_output() to drive it low during sleep, then mux it over to UART TX again when you wake up and maybe even gpio_request/gpio_free as part of this cycle. This all gets very complicated.

The solution is to not think that what the datasheet calls “GPIO mode” has to be handled by the <linux/gpio.h> interface. Instead view this as a certain pin config setting. Look in e.g. <linux/pinctrl/pinconf-generic.h> and you find this in the documentation:

PIN_CONFIG_OUTPUT: this will configure the pin in output, use argument 1 to indicate high level, argument 0 to indicate low level.

So it is perfectly possible to push a pin into “GPIO mode” and drive the line low as part of the usual pin control map. So for example your UART driver may look like this:

```
#include <linux/pinctrl/consumer.h>

struct pinctrl          *pinctrl;
struct pinctrl_state    *pins_default;
struct pinctrl_state    *pins_sleep;

pins_default = pinctrl_lookup_state(uap->pinctrl, PINCTRL_STATE_DEFAULT);
pins_sleep = pinctrl_lookup_state(uap->pinctrl, PINCTRL_STATE_SLEEP);

/* Normal mode */
retval = pinctrl_select_state(pinctrl, pins_default);
/* Sleep mode */
retval = pinctrl_select_state(pinctrl, pins_sleep);
```

And your machine configuration may look like this:

```
static unsigned long uart_default_mode[] = {
        PIN_CONF_PACKED(PIN_CONFIG_DRIVE_PUSH_PULL, 0),
};

static unsigned long uart_sleep_mode[] = {
        PIN_CONF_PACKED(PIN_CONFIG_OUTPUT, 0),
};

static struct pinctrl_map pinmap[] __initdata = {
        PIN_MAP_MUX_GROUP("uart", PINCTRL_STATE_DEFAULT, "pinctrl-foo",
                "u0_group", "u0"),
        PIN_MAP_CONFIGS_PIN("uart", PINCTRL_STATE_DEFAULT, "pinctrl-foo",
                        "UART_TX_PIN", uart_default_mode),
        PIN_MAP_MUX_GROUP("uart", PINCTRL_STATE_SLEEP, "pinctrl-foo",
                "u0_group", "gpio-mode"),
        PIN_MAP_CONFIGS_PIN("uart", PINCTRL_STATE_SLEEP, "pinctrl-foo",
                        "UART_TX_PIN", uart_sleep_mode),
};

foo_init(void) {
        pinctrl_register_mappings(pinmap, ARRAY_SIZE(pinmap));
}
```

Here the pins we want to control are in the “u0_group” and there is some function called “u0” that can be enabled on this group of pins, and then everything is UART business as usual. But there is also some function named “gpio-mode” that can be mapped onto the same pins to move them into GPIO mode.

This will give the desired effect without any bogus interaction with the GPIO subsystem. It is just an electrical configuration used by that device when going to sleep, it might imply that the pin is set into something the datasheet calls “GPIO mode”, but that is not the point: it is still used by that UART device to control the pins that pertain to that very UART driver, putting them into modes needed by the UART. GPIO in the Linux kernel sense are just some 1-bit line, and is a different use case.

How the registers are poked to attain the push or pull, and output low configuration and the muxing of the “u0” or “gpio-mode” group onto these pins is a question for the driver.

Some datasheets will be more helpful and refer to the “GPIO mode” as “low power mode” rather than anything to do with GPIO. This often means the same thing electrically speaking, but in this latter case the software engineers will usually quickly identify that this is some specific muxing or configuration rather than anything related to the GPIO API.

Board/machine configuration

Boards and machines define how a certain complete running system is put together, including how GPIOs and devices are muxed, how regulators are constrained and how the clock tree looks. Of course pinmux settings are also part of this.

A pin controller configuration for a machine looks pretty much like a simple regulator configuration, so for the example array above we want to enable i2c and spi on the second function mapping:

```
#include <linux/pinctrl/machine.h>

static const struct pinctrl_map mapping[] __initconst = {
        {
                .dev_name = "foo-spi.0",
                .name = PINCTRL_STATE_DEFAULT,
                .type = PIN_MAP_TYPE_MUX_GROUP,
                .ctrl_dev_name = "pinctrl-foo",
                .data.mux.function = "spi0",
        },
        {
                .dev_name = "foo-i2c.0",
                .name = PINCTRL_STATE_DEFAULT,
                .type = PIN_MAP_TYPE_MUX_GROUP,
                .ctrl_dev_name = "pinctrl-foo",
                .data.mux.function = "i2c0",
        },
        {
                .dev_name = "foo-mmc.0",
                .name = PINCTRL_STATE_DEFAULT,
                .type = PIN_MAP_TYPE_MUX_GROUP,
                .ctrl_dev_name = "pinctrl-foo",
                .data.mux.function = "mmc0",
        },
};
```

The dev_name here matches to the unique device name that can be used to look up the device struct (just like with clockdev or regulators). The function name must match a function provided by the pinmux driver handling this pin range.

As you can see we may have several pin controllers on the system and thus we need to specify which one of them contains the functions we wish to map.

You register this pinmux mapping to the pinmux subsystem by simply:

```
ret = pinctrl_register_mappings(mapping, ARRAY_SIZE(mapping));
```

Since the above construct is pretty common there is a helper macro to make it even more compact which assumes you want to use pinctrl-foo and position 0 for mapping, for example:

```
static struct pinctrl_map mapping[] __initdata = {
        PIN_MAP_MUX_GROUP("foo-i2c.o", PINCTRL_STATE_DEFAULT,
                          "pinctrl-foo", NULL, "i2c0"),
};
```

The mapping table may also contain pin configuration entries. It's common for each pin/group to have a number of configuration entries that affect it, so the table entries for configuration reference an array of config parameters and values. An example using the convenience macros is shown below:

```
static unsigned long i2c_grp_configs[] = {
        FOO_PIN_DRIVEN,
        FOO_PIN_PULLUP,
};

static unsigned long i2c_pin_configs[] = {
        FOO_OPEN_COLLECTOR,
        FOO_SLEW_RATE_SLOW,
};

static struct pinctrl_map mapping[] __initdata = {
        PIN_MAP_MUX_GROUP("foo-i2c.0", PINCTRL_STATE_DEFAULT,
                          "pinctrl-foo", "i2c0", "i2c0"),
        PIN_MAP_CONFIGS_GROUP("foo-i2c.0", PINCTRL_STATE_DEFAULT,
                              "pinctrl-foo", "i2c0", i2c_grp_configs),
        PIN_MAP_CONFIGS_PIN("foo-i2c.0", PINCTRL_STATE_DEFAULT,
                            "pinctrl-foo", "i2c0scl", i2c_pin_configs),
        PIN_MAP_CONFIGS_PIN("foo-i2c.0", PINCTRL_STATE_DEFAULT,
                            "pinctrl-foo", "i2c0sda", i2c_pin_configs),
};
```

Finally, some devices expect the mapping table to contain certain specific named states. When running on hardware that doesn't need any pin controller configuration, the mapping table must still contain those named states, in order to explicitly indicate that the states were provided and intended to be empty. Table entry macro PIN_MAP_DUMMY_STATE serves the purpose of defining a named state without causing any pin controller to be programmed:

```
static struct pinctrl_map mapping[] __initdata = {
        PIN_MAP_DUMMY_STATE("foo-i2c.0", PINCTRL_STATE_DEFAULT),
};
```

Complex mappings

As it is possible to map a function to different groups of pins an optional .group can be specified like this:

```
...
{
        .dev_name = "foo-spi.0",
        .name = "spi0-pos-A",
        .type = PIN_MAP_TYPE_MUX_GROUP,
        .ctrl_dev_name = "pinctrl-foo",
        .function = "spi0",
        .group = "spi0_0_grp",
},
{
        .dev_name = "foo-spi.0",
        .name = "spi0-pos-B",
        .type = PIN_MAP_TYPE_MUX_GROUP,
        .ctrl_dev_name = "pinctrl-foo",
        .function = "spi0",
        .group = "spi0_1_grp",
},
...
```

This example mapping is used to switch between two positions for spi0 at runtime, as described further below under the heading "Runtime pinmuxing".

Further it is possible for one named state to affect the muxing of several groups of pins, say for example in the mmc0 example above, where you can additively expand the mmc0 bus from 2 to 4 to 8 pins. If we want to use all three groups for a total of 2+2+4 = 8 pins (for an 8-bit MMC bus as is the case), we define a mapping like this:

```
...
{
        .dev_name = "foo-mmc.0",
        .name = "2bit"
        .type = PIN_MAP_TYPE_MUX_GROUP,
        .ctrl_dev_name = "pinctrl-foo",
        .function = "mmc0",
        .group = "mmc0_1_grp",
},
{
        .dev_name = "foo-mmc.0",
        .name = "4bit"
        .type = PIN_MAP_TYPE_MUX_GROUP,
        .ctrl_dev_name = "pinctrl-foo",
        .function = "mmc0",
        .group = "mmc0_1_grp",
},
{
        .dev_name = "foo-mmc.0",
        .name = "4bit"
        .type = PIN_MAP_TYPE_MUX_GROUP,
        .ctrl_dev_name = "pinctrl-foo",
        .function = "mmc0",
        .group = "mmc0_2_grp",
},
{
        .dev_name = "foo-mmc.0",
        .name = "8bit"
        .type = PIN_MAP_TYPE_MUX_GROUP,
        .ctrl_dev_name = "pinctrl-foo",
        .function = "mmc0",
        .group = "mmc0_1_grp",
},
{
        .dev_name = "foo-mmc.0",
        .name = "8bit"
        .type = PIN_MAP_TYPE_MUX_GROUP,
        .ctrl_dev_name = "pinctrl-foo",
        .function = "mmc0",
        .group = "mmc0_2_grp",
},
{
        .dev_name = "foo-mmc.0",
        .name = "8bit"
        .type = PIN_MAP_TYPE_MUX_GROUP,
        .ctrl_dev_name = "pinctrl-foo",
        .function = "mmc0",
        .group = "mmc0_3_grp",
},
...
```

The result of grabbing this mapping from the device with something like this (see next paragraph):

```
p = devm_pinctrl_get(dev);
s = pinctrl_lookup_state(p, "8bit");
ret = pinctrl_select_state(p, s);
```

or more simply:

```
p = devm_pinctrl_get_select(dev, "8bit");
```

Will be that you activate all the three bottom records in the mapping at once. Since they share the same name, pin controller device, function and device, and since we allow multiple groups to match to a single device, they all get selected, and they all get enabled and disable simultaneously by the pinmux core.

Pin control requests from drivers

When a device driver is about to probe the device core will automatically attempt to issue pinctrl_get_select_default() on these devices. This way driver writers do not need to add any of the boilerplate code of the type found below. However when doing fine-grained state selection and not using the "default" state, you may have to do some device driver handling of the pinctrl handles and states.

So if you just want to put the pins for a certain device into the default state and be done with it, there is nothing you need to do besides providing the proper mapping table. The device core will take care of the rest.

Generally it is discouraged to let individual drivers get and enable pin control. So if possible, handle the pin control in platform code or some other place where you have access to all the affected struct device * pointers. In some cases where a driver needs to e.g. switch between different mux mappings at runtime this is not possible.

A typical case is if a driver needs to switch bias of pins from normal operation and going to sleep, moving from the PINCTRL_STATE_DEFAULT to PINCTRL_STATE_SLEEP at runtime, re-biasing or even re-muxing pins to save current in sleep mode.

A driver may request a certain control state to be activated, usually just the default state like this:

```
#include <linux/pinctrl/consumer.h>

struct foo_state {
struct pinctrl *p;
struct pinctrl_state *s;
...
};

foo_probe()
{
        /* Allocate a state holder named "foo" etc */
        struct foo_state *foo = ...;

        foo->p = devm_pinctrl_get(&device);
        if (IS_ERR(foo->p)) {
                /* FIXME: clean up "foo" here */
                return PTR_ERR(foo->p);
        }

        foo->s = pinctrl_lookup_state(foo->p, PINCTRL_STATE_DEFAULT);
        if (IS_ERR(foo->s)) {
                /* FIXME: clean up "foo" here */
                return PTR_ERR(s);
        }

        ret = pinctrl_select_state(foo->s);
        if (ret < 0) {
                /* FIXME: clean up "foo" here */
                return ret;
        }
}
```

This get/lookup/select/put sequence can just as well be handled by bus drivers if you don't want each and every driver to handle it and you know the arrangement on your bus.

The semantics of the pinctrl APIs are:

- pinctrl_get() is called in process context to obtain a handle to all pinctrl information for a given client device. It will allocate a struct from the kernel memory to hold the pinmux state. All mapping table parsing or similar slow operations take place within this API.
- devm_pinctrl_get() is a variant of pinctrl_get() that causes pinctrl_put() to be called automatically on the retrieved pointer when the associated device is removed. It is recommended to use this function over plain pinctrl_get().
- pinctrl_lookup_state() is called in process context to obtain a handle to a specific state for a client device. This operation may be slow, too.
- pinctrl_select_state() programs pin controller hardware according to the definition of the state as given by the mapping table. In theory, this is a fast-path operation, since it only involved blasting some register settings into hardware. However, note that some pin controllers may have their registers on a slow/IRQ-based bus, so client devices should not assume they can call pinctrl_select_state() from non-blocking contexts.
- pinctrl_put() frees all information associated with a pinctrl handle.
- devm_pinctrl_put() is a variant of pinctrl_put() that may be used to explicitly destroy a pinctrl object returned by devm_pinctrl_get(). However, use of this function will be rare, due to the automatic cleanup that will occur even without calling it.

  pinctrl_get() must be paired with a plain pinctrl_put(). pinctrl_get() may not be paired with devm_pinctrl_put(). devm_pinctrl_get() can optionally be paired with devm_pinctrl_put(). devm_pinctrl_get() may not be paired with plain pinctrl_put().

Usually the pin control core handled the get/put pair and call out to the device drivers bookkeeping operations, like checking available functions and the associated pins, whereas select_state pass on to the pin controller driver which takes care of activating and/or deactivating the mux setting by quickly poking some registers.

The pins are allocated for your device when you issue the devm_pinctrl_get() call, after this you should be able to see this in the debugfs listing of all pins.

NOTE: the pinctrl system will return -EPROBE_DEFER if it cannot find the requested pinctrl handles, for example if the pinctrl driver has not yet registered. Thus make sure that the error path in your driver gracefully cleans up and is ready to retry the probing later in the startup process.

Drivers needing both pin control and GPIOs

Again, it is discouraged to let drivers lookup and select pin control states themselves, but again sometimes this is unavoidable.

So say that your driver is fetching its resources like this:

```
#include <linux/pinctrl/consumer.h>
#include <linux/gpio.h>

struct pinctrl *pinctrl;
int gpio;

pinctrl = devm_pinctrl_get_select_default(&dev);
gpio = devm_gpio_request(&dev, 14, "foo");
```

Here we first request a certain pin state and then request GPIO 14 to be used. If you're using the subsystems orthogonally like this, you should nominally always get your pinctrl handle and select the desired pinctrl state BEFORE requesting the GPIO. This is a semantic convention to avoid situations that can be electrically unpleasant, you will certainly want to mux in and bias pins in a certain way before the GPIO subsystems starts to deal with them.

The above can be hidden: using the device core, the pinctrl core may be setting up the config and muxing for the pins right before the device is probing, nevertheless orthogonal to the GPIO subsystem.

But there are also situations where it makes sense for the GPIO subsystem to communicate directly with the pinctrl subsystem, using the latter as a back-end. This is when the GPIO driver may call out to the functions described in the section "Pin control interaction with the GPIO subsystem" above. This only involves per-pin multiplexing, and will be completely hidden behind the gpio_*() function namespace. In this case, the driver need not interact with the pin control subsystem at all.

If a pin control driver and a GPIO driver is dealing with the same pins and the use cases involve multiplexing, you MUST implement the pin controller as a back-end for the GPIO driver like this, unless your hardware design is such that the GPIO controller can override the pin controller's multiplexing state through hardware without the need to interact with the pin control system.

System pin control hogging

Pin control map entries can be hogged by the core when the pin controller is registered. This means that the core will attempt to call pinctrl_get(), lookup_state() and select_state() on it immediately after the pin control device has been registered.

This occurs for mapping table entries where the client device name is equal to the pin controller device name, and the state name is PINCTRL_STATE_DEFAULT:

```
{
        .dev_name = "pinctrl-foo",
        .name = PINCTRL_STATE_DEFAULT,
        .type = PIN_MAP_TYPE_MUX_GROUP,
        .ctrl_dev_name = "pinctrl-foo",
        .function = "power_func",
},
```

Since it may be common to request the core to hog a few always-applicable mux settings on the primary pin controller, there is a convenience macro for this:

```
PIN_MAP_MUX_GROUP_HOG_DEFAULT("pinctrl-foo", NULL /* group */,
                              "power_func")
```

This gives the exact same result as the above construction.

Runtime pinmuxing

It is possible to mux a certain function in and out at runtime, say to move an SPI port from one set of pins to another set of pins. Say for example for spi0 in the example above, we expose two different groups of pins for the same function, but with different named in the mapping as described under "Advanced mapping" above. So that for an SPI device, we have two states named "pos-A" and "pos-B".

This snippet first initializes a state object for both groups (in foo_probe()), then muxes the function in the pins defined by group A, and finally muxes it in on the pins defined by group B:

```
#include <linux/pinctrl/consumer.h>

struct pinctrl *p;
struct pinctrl_state *s1, *s2;

foo_probe()
{
        /* Setup */
        p = devm_pinctrl_get(&device);
        if (IS_ERR(p))
                ...

        s1 = pinctrl_lookup_state(foo->p, "pos-A");
        if (IS_ERR(s1))
                ...

        s2 = pinctrl_lookup_state(foo->p, "pos-B");
        if (IS_ERR(s2))
                ...
}

foo_switch()
{
        /* Enable on position A */
        ret = pinctrl_select_state(s1);
        if (ret < 0)
        ...

        ...

        /* Enable on position B */
        ret = pinctrl_select_state(s2);
        if (ret < 0)
        ...

        ...
}
```

The above has to be done from process context. The reservation of the pins will be done when the state is activated, so in effect one specific pin can be used by different functions at different times on a running system.

CHAPTER

THIRTYFOUR

GENERAL PURPOSE INPUT/OUTPUT (GPIO)

Core

struct **gpio_irq_chip**
GPIO interrupt controller

Definition

```
struct gpio_irq_chip {
  struct irq_chip *chip;
  struct irq_domain *domain;
  const struct irq_domain_ops *domain_ops;
  irq_flow_handler_t handler;
  unsigned int default_type;
  struct lock_class_key *lock_key;
  struct lock_class_key *request_key;
  irq_flow_handler_t parent_handler;
  void *parent_handler_data;
  unsigned int num_parents;
  unsigned int *parents;
  unsigned int *map;
  bool threaded;
  bool need_valid_mask;
  unsigned long *valid_mask;
  unsigned int first;
};
```

Members

chip GPIO IRQ chip implementation, provided by GPIO driver.

domain Interrupt translation domain; responsible for mapping between GPIO hwirq number and Linux IRQ number.

domain_ops Table of interrupt domain operations for this IRQ chip.

handler The IRQ handler to use (often a predefined IRQ core function) for GPIO IRQs, provided by GPIO driver.

default_type Default IRQ triggering type applied during GPIO driver initialization, provided by GPIO driver.

lock_key Per GPIO IRQ chip lockdep classes.

parent_handler The interrupt handler for the GPIO chip's parent interrupts, may be NULL if the parent interrupts are nested rather than cascaded.

parent_handler_data Data associated, and passed to, the handler for the parent interrupt.

num_parents The number of interrupt parents of a GPIO chip.

parents A list of interrupt parents of a GPIO chip. This is owned by the driver, so the core will only reference this list, not modify it.

map A list of interrupt parents for each line of a GPIO chip.

threaded True if set the interrupt handling uses nested threads.

need_valid_mask If set core allocates **valid_mask** with all bits set to one.

valid_mask If not NULL holds bitmask of GPIOs which are valid to be included in IRQ domain of the chip.

first Required for static IRQ allocation. If set, irq_domain_add_simple() will allocate and map all IRQs during initialization.

struct **gpio_chip**
abstract a GPIO controller

Definition

```
struct gpio_chip {
  const char              *label;
  struct gpio_device      *gpiodev;
  struct device           *parent;
  struct module           *owner;
  int (*request)(struct gpio_chip *chip, unsigned offset);
  void (*free)(struct gpio_chip *chip, unsigned offset);
  int (*get_direction)(struct gpio_chip *chip, unsigned offset);
  int (*direction_input)(struct gpio_chip *chip, unsigned offset);
  int (*direction_output)(struct gpio_chip *chip, unsigned offset, int value);
  int (*get)(struct gpio_chip *chip, unsigned offset);
  int (*get_multiple)(struct gpio_chip *chip,unsigned long *mask, unsigned long *bits);
  void (*set)(struct gpio_chip *chip, unsigned offset, int value);
  void (*set_multiple)(struct gpio_chip *chip,unsigned long *mask, unsigned long *bits);
  int (*set_config)(struct gpio_chip *chip,unsigned offset, unsigned long config);
  int (*to_irq)(struct gpio_chip *chip, unsigned offset);
  void (*dbg_show)(struct seq_file *s, struct gpio_chip *chip);
  int base;
  u16 ngpio;
  const char              *const *names;
  bool can_sleep;
#if IS_ENABLED(CONFIG_GPIO_GENERIC);
  unsigned long (*read_reg)(void __iomem *reg);
  void (*write_reg)(void __iomem *reg, unsigned long data);
  bool be_bits;
  void __iomem *reg_dat;
  void __iomem *reg_set;
  void __iomem *reg_clr;
  void __iomem *reg_dir;
  int bgpio_bits;
  spinlock_t bgpio_lock;
  unsigned long bgpio_data;
  unsigned long bgpio_dir;
#endif;
#ifdef CONFIG_GPIOLIB_IRQCHIP;
  struct gpio_irq_chip irq;
#endif;
#if defined(CONFIG_OF_GPIO);
  struct device_node *of_node;
  unsigned int of_gpio_n_cells;
  int (*of_xlate)(struct gpio_chip *gc, const struct of_phandle_args *gpiospec, u32 *flags);
#endif;
};
```

Members

label a functional name for the GPIO device, such as a part number or the name of the SoC IP-block implementing it.

gpiodev the internal state holder, opaque struct

parent optional parent device providing the GPIOs

owner helps prevent removal of modules exporting active GPIOs

request optional hook for chip-specific activation, such as enabling module power and clock; may sleep

free optional hook for chip-specific deactivation, such as disabling module power and clock; may sleep

get_direction returns direction for signal "offset", 0=out, 1=in, (same as GPIOF_DIR_XXX), or negative error

direction_input configures signal "offset" as input, or returns error

direction_output configures signal "offset" as output, or returns error

get returns value for signal "offset", 0=low, 1=high, or negative error

get_multiple reads values for multiple signals defined by "mask" and stores them in "bits", returns 0 on success or negative error

set assigns output value for signal "offset"

set_multiple assigns output values for multiple signals defined by "mask"

set_config optional hook for all kinds of settings. Uses the same packed config format as generic pinconf.

to_irq optional hook supporting non-static `gpio_to_irq()` mappings; implementation may not sleep

dbg_show optional routine to show contents in debugfs; default code will be used when this is omitted, but custom code can show extra state (such as pullup/pulldown configuration).

base identifies the first GPIO number handled by this chip; or, if negative during registration, requests dynamic ID allocation. DEPRECATION: providing anything non-negative and nailing the base offset of GPIO chips is deprecated. Please pass -1 as base to let gpiolib select the chip base in all possible cases. We want to get rid of the static GPIO number space in the long run.

ngpio the number of GPIOs handled by this controller; the last GPIO handled is (base + ngpio - 1).

names if set, must be an array of strings to use as alternative names for the GPIOs in this chip. Any entry in the array may be NULL if there is no alias for the GPIO, however the array must be **ngpio** entries long. A name can include a single printk format specifier for an unsigned int. It is substituted by the actual number of the gpio.

can_sleep flag must be set iff `get()`/`set()` methods sleep, as they must while accessing GPIO expander chips over I2C or SPI. This implies that if the chip supports IRQs, these IRQs need to be threaded as the chip access may sleep when e.g. reading out the IRQ status registers.

read_reg reader function for generic GPIO

write_reg writer function for generic GPIO

be_bits if the generic GPIO has big endian bit order (bit 31 is representing line 0, bit 30 is line 1 ... bit 0 is line 31) this is set to true by the generic GPIO core. It is for internal housekeeping only.

reg_dat data (in) register for generic GPIO

reg_set output set register (out=high) for generic GPIO

reg_clr output clear register (out=low) for generic GPIO

reg_dir direction setting register for generic GPIO

bgpio_bits number of register bits used for a generic GPIO i.e. <register width> * 8

bgpio_lock used to lock chip->bgpio_data. Also, this is needed to keep shadowed and real data registers writes together.

bgpio_data shadowed data register for generic GPIO to clear/set bits safely.

bgpio_dir shadowed direction register for generic GPIO to clear/set direction safely.

irq Integrates interrupt chip functionality with the GPIO chip. Can be used to handle IRQs for most practical cases.

of_node Pointer to a device tree node representing this GPIO controller.

of_gpio_n_cells Number of cells used to form the GPIO specifier.

of_xlate Callback to translate a device tree GPIO specifier into a chip- relative GPIO number and flags.

Description

A gpio_chip can help platforms abstract various sources of GPIOs so they can all be accessed through a common programing interface. Example sources would be SOC controllers, FPGAs, multifunction chips, dedicated GPIO expanders, and so on.

Each chip controls a number of signals, identified in method calls by "offset" values in the range 0..(**ngpio** - 1). When those signals are referenced through calls like gpio_get_value(gpio), the offset is calculated by subtracting **base** from the gpio number.

gpiochip_add_data(*chip*, *data*)
register a gpio_chip

Parameters

chip the chip to register, with chip->base initialized

data driver-private data associated with this chip

Context

potentially before irqs will work

Description

When *gpiochip_add_data()* is called very early during boot, so that GPIOs can be freely used, the chip->parent device must be registered before the gpio framework's `arch_initcall()`. Otherwise sysfs initialization for GPIOs will fail rudely.

gpiochip_add_data() must only be called after gpiolib initialization, ie after `core_initcall()`.

If chip->base is negative, this requests dynamic assignment of a range of valid GPIOs.

Return

A negative errno if the chip can't be registered, such as because the chip->base is invalid or already associated with a different chip. Otherwise it returns zero as a success code.

struct **gpio_pin_range**
pin range controlled by a gpio chip

Definition

```
struct gpio_pin_range {
  struct list_head node;
  struct pinctrl_dev *pctldev;
  struct pinctrl_gpio_range range;
};
```

Members

node list for maintaining set of pin ranges, used internally

pctldev pinctrl device which handles corresponding pins

range actual range of pins controlled by a gpio controller

struct gpio_desc * **gpio_to_desc**(unsigned *gpio*)
Convert a GPIO number to its descriptor

Parameters

unsigned gpio global GPIO number

Return

The GPIO descriptor associated with the given GPIO, or `NULL` if no GPIO with the given number exists in the system.

int **`desc_to_gpio`**(const struct gpio_desc * *desc*)
convert a GPIO descriptor to the integer namespace

Parameters

`const struct gpio_desc * desc` GPIO descriptor

Description

This should disappear in the future but is needed since we still use GPIO numbers for error messages and sysfs nodes.

Return

The global GPIO number for the GPIO specified by its descriptor.

struct *gpio_chip* * **`gpiod_to_chip`**(const struct gpio_desc * *desc*)
Return the GPIO chip to which a GPIO descriptor belongs

Parameters

`const struct gpio_desc * desc` descriptor to return the chip of

int **`gpiod_get_direction`**(struct gpio_desc * *desc*)
return the current direction of a GPIO

Parameters

`struct gpio_desc * desc` GPIO to get the direction of

Description

Returns 0 for output, 1 for input, or an error code in case of error.

This function may sleep if *`gpiod_cansleep()`* is true.

void * **`gpiochip_get_data`**(struct *gpio_chip* * *chip*)
get per-subdriver data for the chip

Parameters

`struct gpio_chip * chip` GPIO chip

Return

The per-subdriver data for the chip.

void **`gpiochip_remove`**(struct *gpio_chip* * *chip*)
unregister a gpio_chip

Parameters

`struct gpio_chip * chip` the chip to unregister

Description

A gpio_chip with any GPIOs still requested may not be removed.

int **`devm_gpiochip_add_data`**(struct *device* * *dev*, struct *gpio_chip* * *chip*, void * *data*)
Resource manager *`gpiochip_add_data()`*

Parameters

`struct device * dev` the device pointer on which irq_chip belongs to.

`struct gpio_chip * chip` the chip to register, with chip->base initialized

`void * data` driver-private data associated with this chip

Context

potentially before irqs will work

Description

The gpio chip automatically be released when the device is unbound.

Return

A negative errno if the chip can't be registered, such as because the chip->base is invalid or already associated with a different chip. Otherwise it returns zero as a success code.

void **devm_gpiochip_remove**(struct *device* * *dev*, struct *gpio_chip* * *chip*)
Resource manager of *gpiochip_remove()*

Parameters

struct device * dev device for which which resource was allocated

struct gpio_chip * chip the chip to remove

Description

A gpio_chip with any GPIOs still requested may not be removed.

struct *gpio_chip* * **gpiochip_find**(void * *data*, int (*match) (struct *gpio_chip* **chip*, void **data*)
iterator for locating a specific gpio_chip

Parameters

void * data data to pass to match function

int (*)(struct gpio_chip *chip, void *data) match Callback function to check gpio_chip

Description

Similar to bus_find_device. It returns a reference to a gpio_chip as determined by a user supplied **match** callback. The callback should return 0 if the device doesn't match and non-zero if it does. If the callback is non-zero, this function will return to the caller and not iterate over any more gpio_chips.

void **gpiochip_set_chained_irqchip**(struct *gpio_chip* * *gpiochip*, struct irq_chip * *irqchip*, unsigned int *parent_irq*, irq_flow_handler_t *parent_handler*)
connects a chained irqchip to a gpiochip

Parameters

struct gpio_chip * gpiochip the gpiochip to set the irqchip chain to

struct irq_chip * irqchip the irqchip to chain to the gpiochip

unsigned int parent_irq the irq number corresponding to the parent IRQ for this chained irqchip

irq_flow_handler_t parent_handler the parent interrupt handler for the accumulated IRQ coming out of the gpiochip. If the interrupt is nested rather than cascaded, pass NULL in this handler argument

void **gpiochip_set_nested_irqchip**(struct *gpio_chip* * *gpiochip*, struct irq_chip * *irqchip*, unsigned int *parent_irq*)
connects a nested irqchip to a gpiochip

Parameters

struct gpio_chip * gpiochip the gpiochip to set the irqchip nested handler to

struct irq_chip * irqchip the irqchip to nest to the gpiochip

unsigned int parent_irq the irq number corresponding to the parent IRQ for this nested irqchip

int **gpiochip_irq_map**(struct irq_domain * *d*, unsigned int *irq*, irq_hw_number_t *hwirq*)
maps an IRQ into a GPIO irqchip

Parameters

struct irq_domain * d the irqdomain used by this irqchip

unsigned int irq the global irq number used by this GPIO irqchip irq

irq_hw_number_t hwirq the local IRQ/GPIO line offset on this gpiochip

Description

This function will set up the mapping for a certain IRQ line on a gpiochip by assigning the gpiochip as chip data, and using the irqchip stored inside the gpiochip.

int **gpiochip_irqchip_add_key**(struct *gpio_chip* * *gpiochip*, struct irq_chip * *irqchip*, unsigned int *first_irq*, irq_flow_handler_t *handler*, unsigned int *type*, bool *threaded*, struct lock_class_key * *lock_key*, struct lock_class_key * *request_key*)
adds an irqchip to a gpiochip

Parameters

struct gpio_chip * gpiochip the gpiochip to add the irqchip to

struct irq_chip * irqchip the irqchip to add to the gpiochip

unsigned int first_irq if not dynamically assigned, the base (first) IRQ to allocate gpiochip irqs from

irq_flow_handler_t handler the irq handler to use (often a predefined irq core function)

unsigned int type the default type for IRQs on this irqchip, pass IRQ_TYPE_NONE to have the core avoid setting up any default type in the hardware.

bool threaded whether this irqchip uses a nested thread handler

struct lock_class_key * lock_key lockdep class for IRQ lock

struct lock_class_key * request_key lockdep class for IRQ request

Description

This function closely associates a certain irqchip with a certain gpiochip, providing an irq domain to translate the local IRQs to global irqs in the gpiolib core, and making sure that the gpiochip is passed as chip data to all related functions. Driver callbacks need to use `gpiochip_get_data()` to get their local state containers back from the gpiochip passed as chip data. An irqdomain will be stored in the gpiochip that shall be used by the driver to handle IRQ number translation. The gpiochip will need to be initialized and registered before calling this function.

This function will handle two cell:ed simple IRQs and assumes all the pins on the gpiochip can generate a unique IRQ. Everything else need to be open coded.

int **gpiochip_generic_request**(struct *gpio_chip* * *chip*, unsigned *offset*)
request the gpio function for a pin

Parameters

struct gpio_chip * chip the gpiochip owning the GPIO

unsigned offset the offset of the GPIO to request for GPIO function

void **gpiochip_generic_free**(struct *gpio_chip* * *chip*, unsigned *offset*)
free the gpio function from a pin

Parameters

struct gpio_chip * chip the gpiochip to request the gpio function for

unsigned offset the offset of the GPIO to free from GPIO function

int **gpiochip_generic_config**(struct *gpio_chip* * *chip*, unsigned *offset*, unsigned long *config*)
apply configuration for a pin

Parameters

struct gpio_chip * chip the gpiochip owning the GPIO

unsigned offset the offset of the GPIO to apply the configuration

unsigned long config the configuration to be applied

int **gpiochip_add_pingroup_range**(struct *gpio_chip* * *chip*, struct pinctrl_dev * *pctldev*, unsigned int *gpio_offset*, const char * *pin_group*)
add a range for GPIO <-> pin mapping

Parameters

struct gpio_chip * chip the gpiochip to add the range for

struct pinctrl_dev * pctldev the pin controller to map to

unsigned int gpio_offset the start offset in the current gpio_chip number space

const char * pin_group name of the pin group inside the pin controller

int **gpiochip_add_pin_range**(struct *gpio_chip* * *chip*, const char * *pinctl_name*, unsigned int *gpio_offset*, unsigned int *pin_offset*, unsigned int *npins*)
add a range for GPIO <-> pin mapping

Parameters

struct gpio_chip * chip the gpiochip to add the range for

const char * pinctl_name the dev_name() of the pin controller to map to

unsigned int gpio_offset the start offset in the current gpio_chip number space

unsigned int pin_offset the start offset in the pin controller number space

unsigned int npins the number of pins from the offset of each pin space (GPIO and pin controller) to accumulate in this range

Return

0 on success, or a negative error-code on failure.

void **gpiochip_remove_pin_ranges**(struct *gpio_chip* * *chip*)
remove all the GPIO <-> pin mappings

Parameters

struct gpio_chip * chip the chip to remove all the mappings for

const char * **gpiochip_is_requested**(struct *gpio_chip* * *chip*, unsigned *offset*)
return string iff signal was requested

Parameters

struct gpio_chip * chip controller managing the signal

unsigned offset of signal within controller's 0..(ngpio - 1) range

Description

Returns NULL if the GPIO is not currently requested, else a string. The string returned is the label passed to gpio_request(); if none has been passed it is a meaningless, non-NULL constant.

This function is for use by GPIO controller drivers. The label can help with diagnostics, and knowing that the signal is used as a GPIO can help avoid accidentally multiplexing it to another controller.

struct gpio_desc * **gpiochip_request_own_desc**(struct *gpio_chip* * *chip*, u16 *hwnum*, const char * *label*)
Allow GPIO chip to request its own descriptor

Parameters

struct gpio_chip * chip GPIO chip

u16 hwnum hardware number of the GPIO for which to request the descriptor

const char * label label for the GPIO

Description

Function allows GPIO chip drivers to request and use their own GPIO descriptors via gpiolib API. Difference to `gpiod_request()` is that this function will not increase reference count of the GPIO chip module. This allows the GPIO chip module to be unloaded as needed (we assume that the GPIO chip driver handles freeing the GPIOs it has requested).

Return

A pointer to the GPIO descriptor, or an `ERR_PTR()`-encoded negative error code on failure.

void **`gpiochip_free_own_desc`**(struct gpio_desc * *desc*)
Free GPIO requested by the chip driver

Parameters

`struct gpio_desc * desc` GPIO descriptor to free

Description

Function frees the given GPIO requested previously with *`gpiochip_request_own_desc()`*.

int **`gpiod_direction_input`**(struct gpio_desc * *desc*)
set the GPIO direction to input

Parameters

`struct gpio_desc * desc` GPIO to set to input

Description

Set the direction of the passed GPIO to input, such as *`gpiod_get_value()`* can be called safely on it.

Return 0 in case of success, else an error code.

int **`gpiod_direction_output_raw`**(struct gpio_desc * *desc*, int *value*)
set the GPIO direction to output

Parameters

`struct gpio_desc * desc` GPIO to set to output

`int value` initial output value of the GPIO

Description

Set the direction of the passed GPIO to output, such as *`gpiod_set_value()`* can be called safely on it. The initial value of the output must be specified as raw value on the physical line without regard for the ACTIVE_LOW status.

Return 0 in case of success, else an error code.

int **`gpiod_direction_output`**(struct gpio_desc * *desc*, int *value*)
set the GPIO direction to output

Parameters

`struct gpio_desc * desc` GPIO to set to output

`int value` initial output value of the GPIO

Description

Set the direction of the passed GPIO to output, such as *`gpiod_set_value()`* can be called safely on it. The initial value of the output must be specified as the logical value of the GPIO, i.e. taking its ACTIVE_LOW status into account.

Return 0 in case of success, else an error code.

int **`gpiod_set_debounce`**(struct gpio_desc * *desc*, unsigned *debounce*)
sets **debounce** time for a GPIO

Parameters

struct gpio_desc * desc descriptor of the GPIO for which to set debounce time

unsigned debounce debounce time in microseconds

Return

0 on success, -ENOTSUPP if the controller doesn't support setting the debounce time.

int **gpiod_set_transitory**(struct gpio_desc * *desc*, bool *transitory*)
Lose or retain GPIO state on suspend or reset

Parameters

struct gpio_desc * desc descriptor of the GPIO for which to configure persistence

bool transitory True to lose state on suspend or reset, false for persistence

Return

0 on success, otherwise a negative error code.

int **gpiod_is_active_low**(const struct gpio_desc * *desc*)
test whether a GPIO is active-low or not

Parameters

const struct gpio_desc * desc the gpio descriptor to test

Description

Returns 1 if the GPIO is active-low, 0 otherwise.

int **gpiod_get_raw_value**(const struct gpio_desc * *desc*)
return a gpio's raw value

Parameters

const struct gpio_desc * desc gpio whose value will be returned

Description

Return the GPIO's raw value, i.e. the value of the physical line disregarding its ACTIVE_LOW status, or negative errno on failure.

This function should be called from contexts where we cannot sleep, and will complain if the GPIO chip functions potentially sleep.

int **gpiod_get_value**(const struct gpio_desc * *desc*)
return a gpio's value

Parameters

const struct gpio_desc * desc gpio whose value will be returned

Description

Return the GPIO's logical value, i.e. taking the ACTIVE_LOW status into account, or negative errno on failure.

This function should be called from contexts where we cannot sleep, and will complain if the GPIO chip functions potentially sleep.

int **gpiod_get_raw_array_value**(unsigned int *array_size*, struct gpio_desc ** *desc_array*, int * *value_array*)
read raw values from an array of GPIOs

Parameters

unsigned int array_size number of elements in the descriptor / value arrays

struct gpio_desc ** desc_array array of GPIO descriptors whose values will be read

int * value_array array to store the read values

Description

Read the raw values of the GPIOs, i.e. the values of the physical lines without regard for their ACTIVE_LOW status. Return 0 in case of success, else an error code.

This function should be called from contexts where we cannot sleep, and it will complain if the GPIO chip functions potentially sleep.

int **`gpiod_get_array_value`**(unsigned int *array_size*, struct gpio_desc ** *desc_array*, int * *value_array*)
read values from an array of GPIOs

Parameters

`unsigned int array_size` number of elements in the descriptor / value arrays

`struct gpio_desc ** desc_array` array of GPIO descriptors whose values will be read

`int * value_array` array to store the read values

Description

Read the logical values of the GPIOs, i.e. taking their ACTIVE_LOW status into account. Return 0 in case of success, else an error code.

This function should be called from contexts where we cannot sleep, and it will complain if the GPIO chip functions potentially sleep.

void **`gpiod_set_raw_value`**(struct gpio_desc * *desc*, int *value*)
assign a gpio's raw value

Parameters

`struct gpio_desc * desc` gpio whose value will be assigned

`int value` value to assign

Description

Set the raw value of the GPIO, i.e. the value of its physical line without regard for its ACTIVE_LOW status.

This function should be called from contexts where we cannot sleep, and will complain if the GPIO chip functions potentially sleep.

void **`gpiod_set_value`**(struct gpio_desc * *desc*, int *value*)
assign a gpio's value

Parameters

`struct gpio_desc * desc` gpio whose value will be assigned

`int value` value to assign

Description

Set the logical value of the GPIO, i.e. taking its ACTIVE_LOW, OPEN_DRAIN and OPEN_SOURCE flags into account.

This function should be called from contexts where we cannot sleep, and will complain if the GPIO chip functions potentially sleep.

void **`gpiod_set_raw_array_value`**(unsigned int *array_size*, struct gpio_desc ** *desc_array*, int * *value_array*)
assign values to an array of GPIOs

Parameters

`unsigned int array_size` number of elements in the descriptor / value arrays

`struct gpio_desc ** desc_array` array of GPIO descriptors whose values will be assigned

`int * value_array` array of values to assign

Description

Set the raw values of the GPIOs, i.e. the values of the physical lines without regard for their ACTIVE_LOW status.

This function should be called from contexts where we cannot sleep, and will complain if the GPIO chip functions potentially sleep.

void **gpiod_set_array_value**(unsigned int *array_size*, struct gpio_desc ** *desc_array*, int * *value_array*)
assign values to an array of GPIOs

Parameters

unsigned int array_size number of elements in the descriptor / value arrays

struct gpio_desc ** desc_array array of GPIO descriptors whose values will be assigned

int * value_array array of values to assign

Description

Set the logical values of the GPIOs, i.e. taking their ACTIVE_LOW status into account.

This function should be called from contexts where we cannot sleep, and will complain if the GPIO chip functions potentially sleep.

int **gpiod_cansleep**(const struct gpio_desc * *desc*)
report whether gpio value access may sleep

Parameters

const struct gpio_desc * desc gpio to check

int **gpiod_to_irq**(const struct gpio_desc * *desc*)
return the IRQ corresponding to a GPIO

Parameters

const struct gpio_desc * desc gpio whose IRQ will be returned (already requested)

Description

Return the IRQ corresponding to the passed GPIO, or an error code in case of error.

int **gpiochip_lock_as_irq**(struct *gpio_chip* * *chip*, unsigned int *offset*)
lock a GPIO to be used as IRQ

Parameters

struct gpio_chip * chip the chip the GPIO to lock belongs to

unsigned int offset the offset of the GPIO to lock as IRQ

Description

This is used directly by GPIO drivers that want to lock down a certain GPIO line to be used for IRQs.

void **gpiochip_unlock_as_irq**(struct *gpio_chip* * *chip*, unsigned int *offset*)
unlock a GPIO used as IRQ

Parameters

struct gpio_chip * chip the chip the GPIO to lock belongs to

unsigned int offset the offset of the GPIO to lock as IRQ

Description

This is used directly by GPIO drivers that want to indicate that a certain GPIO is no longer used exclusively for IRQ.

int **gpiod_get_raw_value_cansleep**(const struct gpio_desc * *desc*)
return a gpio's raw value

Parameters

`const struct gpio_desc * desc` gpio whose value will be returned

Description

Return the GPIO's raw value, i.e. the value of the physical line disregarding its ACTIVE_LOW status, or negative errno on failure.

This function is to be called from contexts that can sleep.

int **`gpiod_get_value_cansleep`**(const struct gpio_desc * *desc*)
return a gpio's value

Parameters

`const struct gpio_desc * desc` gpio whose value will be returned

Description

Return the GPIO's logical value, i.e. taking the ACTIVE_LOW status into account, or negative errno on failure.

This function is to be called from contexts that can sleep.

int **`gpiod_get_raw_array_value_cansleep`**(unsigned int *array_size*, struct gpio_desc ** *desc_array*, int * *value_array*)
read raw values from an array of GPIOs

Parameters

`unsigned int array_size` number of elements in the descriptor / value arrays

`struct gpio_desc ** desc_array` array of GPIO descriptors whose values will be read

`int * value_array` array to store the read values

Description

Read the raw values of the GPIOs, i.e. the values of the physical lines without regard for their ACTIVE_LOW status. Return 0 in case of success, else an error code.

This function is to be called from contexts that can sleep.

int **`gpiod_get_array_value_cansleep`**(unsigned int *array_size*, struct gpio_desc ** *desc_array*, int * *value_array*)
read values from an array of GPIOs

Parameters

`unsigned int array_size` number of elements in the descriptor / value arrays

`struct gpio_desc ** desc_array` array of GPIO descriptors whose values will be read

`int * value_array` array to store the read values

Description

Read the logical values of the GPIOs, i.e. taking their ACTIVE_LOW status into account. Return 0 in case of success, else an error code.

This function is to be called from contexts that can sleep.

void **`gpiod_set_raw_value_cansleep`**(struct gpio_desc * *desc*, int *value*)
assign a gpio's raw value

Parameters

`struct gpio_desc * desc` gpio whose value will be assigned

`int value` value to assign

Description

Set the raw value of the GPIO, i.e. the value of its physical line without regard for its ACTIVE_LOW status.

This function is to be called from contexts that can sleep.

void **gpiod_set_value_cansleep**(struct gpio_desc * *desc*, int *value*)
assign a gpio's value

Parameters

struct gpio_desc * desc gpio whose value will be assigned

int value value to assign

Description

Set the logical value of the GPIO, i.e. taking its ACTIVE_LOW status into account

This function is to be called from contexts that can sleep.

void **gpiod_set_raw_array_value_cansleep**(unsigned int *array_size*, struct gpio_desc ** *desc_array*, int * *value_array*)
assign values to an array of GPIOs

Parameters

unsigned int array_size number of elements in the descriptor / value arrays

struct gpio_desc ** desc_array array of GPIO descriptors whose values will be assigned

int * value_array array of values to assign

Description

Set the raw values of the GPIOs, i.e. the values of the physical lines without regard for their ACTIVE_LOW status.

This function is to be called from contexts that can sleep.

void **gpiod_set_array_value_cansleep**(unsigned int *array_size*, struct gpio_desc ** *desc_array*, int * *value_array*)
assign values to an array of GPIOs

Parameters

unsigned int array_size number of elements in the descriptor / value arrays

struct gpio_desc ** desc_array array of GPIO descriptors whose values will be assigned

int * value_array array of values to assign

Description

Set the logical values of the GPIOs, i.e. taking their ACTIVE_LOW status into account.

This function is to be called from contexts that can sleep.

void **gpiod_add_lookup_table**(struct gpiod_lookup_table * *table*)
register GPIO device consumers

Parameters

struct gpiod_lookup_table * table table of consumers to register

void **gpiod_remove_lookup_table**(struct gpiod_lookup_table * *table*)
unregister GPIO device consumers

Parameters

struct gpiod_lookup_table * table table of consumers to unregister

int **gpiod_count**(struct *device* * *dev*, const char * *con_id*)
return the number of GPIOs associated with a device / function or -ENOENT if no GPIO has been assigned to the requested function

Parameters

struct device * dev GPIO consumer, can be NULL for system-global GPIOs

const char * con_id function within the GPIO consumer

struct gpio_desc * **gpiod_get**(struct *device* * *dev*, const char * *con_id*, enum gpiod_flags *flags*)
obtain a GPIO for a given GPIO function

Parameters

struct device * dev GPIO consumer, can be NULL for system-global GPIOs

const char * con_id function within the GPIO consumer

enum gpiod_flags flags optional GPIO initialization flags

Description

Return the GPIO descriptor corresponding to the function con_id of device dev, -ENOENT if no GPIO has been assigned to the requested function, or another `IS_ERR()` code if an error occurred while trying to acquire the GPIO.

struct gpio_desc * **gpiod_get_optional**(struct *device* * *dev*, const char * *con_id*, enum gpiod_flags *flags*)
obtain an optional GPIO for a given GPIO function

Parameters

struct device * dev GPIO consumer, can be NULL for system-global GPIOs

const char * con_id function within the GPIO consumer

enum gpiod_flags flags optional GPIO initialization flags

Description

This is equivalent to *gpiod_get()*, except that when no GPIO was assigned to the requested function it will return NULL. This is convenient for drivers that need to handle optional GPIOs.

struct gpio_desc * **gpiod_get_index**(struct *device* * *dev*, const char * *con_id*, unsigned int *idx*, enum gpiod_flags *flags*)
obtain a GPIO from a multi-index GPIO function

Parameters

struct device * dev GPIO consumer, can be NULL for system-global GPIOs

const char * con_id function within the GPIO consumer

unsigned int idx index of the GPIO to obtain in the consumer

enum gpiod_flags flags optional GPIO initialization flags

Description

This variant of *gpiod_get()* allows to access GPIOs other than the first defined one for functions that define several GPIOs.

Return a valid GPIO descriptor, -ENOENT if no GPIO has been assigned to the requested function and/or index, or another `IS_ERR()` code if an error occurred while trying to acquire the GPIO.

struct gpio_desc * **gpiod_get_from_of_node**(struct device_node * *node*, const char * *propname*, int *index*, enum gpiod_flags *dflags*, const char * *label*)
obtain a GPIO from an OF node

Parameters

struct device_node * node handle of the OF node

const char * propname name of the DT property representing the GPIO

int index index of the GPIO to obtain for the consumer

enum gpiod_flags dflags GPIO initialization flags

const char * label label to attach to the requested GPIO

Return

On successful request the GPIO pin is configured in accordance with provided **dflags**. If the node does not have the requested GPIO property, NULL is returned.

In case of error an ERR_PTR() is returned.

struct gpio_desc * **fwnode_get_named_gpiod**(struct fwnode_handle * *fwnode*, const char * *propname*, int *index*, enum gpiod_flags *dflags*, const char * *label*)

obtain a GPIO from firmware node

Parameters

struct fwnode_handle * fwnode handle of the firmware node

const char * propname name of the firmware property representing the GPIO

int index index of the GPIO to obtain for the consumer

enum gpiod_flags dflags GPIO initialization flags

const char * label label to attach to the requested GPIO

Description

This function can be used for drivers that get their configuration from opaque firmware.

The function properly finds the corresponding GPIO using whatever is the underlying firmware interface and then makes sure that the GPIO descriptor is requested before it is returned to the caller.

Return

On successful request the GPIO pin is configured in accordance with provided **dflags**.

In case of error an ERR_PTR() is returned.

struct gpio_desc * **gpiod_get_index_optional**(struct *device* * *dev*, const char * *con_id*, unsigned int *index*, enum gpiod_flags *flags*)

obtain an optional GPIO from a multi-index GPIO function

Parameters

struct device * dev GPIO consumer, can be NULL for system-global GPIOs

const char * con_id function within the GPIO consumer

unsigned int index index of the GPIO to obtain in the consumer

enum gpiod_flags flags optional GPIO initialization flags

Description

This is equivalent to *gpiod_get_index()*, except that when no GPIO with the specified index was assigned to the requested function it will return NULL. This is convenient for drivers that need to handle optional GPIOs.

struct gpio_descs * **gpiod_get_array**(struct *device* * *dev*, const char * *con_id*, enum gpiod_flags *flags*)

obtain multiple GPIOs from a multi-index GPIO function

Parameters

struct device * dev GPIO consumer, can be NULL for system-global GPIOs

const char * con_id function within the GPIO consumer

`enum gpiod_flags flags` optional GPIO initialization flags

Description

This function acquires all the GPIOs defined under a given function.

Return a struct gpio_descs containing an array of descriptors, -ENOENT if no GPIO has been assigned to the requested function, or another `IS_ERR()` code if an error occurred while trying to acquire the GPIOs.

struct gpio_descs * **`gpiod_get_array_optional`**(struct *device* * *dev*, const char * *con_id*, enum gpiod_flags *flags*)
obtain multiple GPIOs from a multi-index GPIO function

Parameters

`struct device * dev` GPIO consumer, can be NULL for system-global GPIOs

`const char * con_id` function within the GPIO consumer

`enum gpiod_flags flags` optional GPIO initialization flags

Description

This is equivalent to *`gpiod_get_array()`*, except that when no GPIO was assigned to the requested function it will return NULL.

void **`gpiod_put`**(struct gpio_desc * *desc*)
dispose of a GPIO descriptor

Parameters

`struct gpio_desc * desc` GPIO descriptor to dispose of

Description

No descriptor can be used after *`gpiod_put()`* has been called on it.

void **`gpiod_put_array`**(struct gpio_descs * *descs*)
dispose of multiple GPIO descriptors

Parameters

`struct gpio_descs * descs` struct gpio_descs containing an array of descriptors

Legacy API

The functions listed in this section are deprecated. The GPIO descriptor based API described above should be used in new code.

int **`gpio_request_one`**(unsigned *gpio*, unsigned long *flags*, const char * *label*)
request a single GPIO with initial configuration

Parameters

`unsigned gpio` the GPIO number

`unsigned long flags` GPIO configuration as specified by GPIOF_*

`const char * label` a literal description string of this GPIO

int **`gpio_request_array`**(const struct gpio * *array*, size_t *num*)
request multiple GPIOs in a single call

Parameters

`const struct gpio * array` array of the 'struct gpio'

`size_t num` how many GPIOs in the array

void **gpio_free_array**(const struct gpio * *array*, size_t *num*)
release multiple GPIOs in a single call

Parameters

const struct gpio * array array of the 'struct gpio'

size_t num how many GPIOs in the array

ACPI support

void **acpi_gpiochip_request_interrupts**(struct *gpio_chip* * *chip*)
Register isr for gpio chip ACPI events

Parameters

struct gpio_chip * chip GPIO chip

Description

ACPI5 platforms can use GPIO signaled ACPI events. These GPIO interrupts are handled by ACPI event methods which need to be called from the GPIO chip's interrupt handler. acpi_gpiochip_request_interrupts finds out which gpio pins have acpi event methods and assigns interrupt handlers that calls the acpi event methods for those pins.

void **acpi_gpiochip_free_interrupts**(struct *gpio_chip* * *chip*)
Free GPIO ACPI event interrupts.

Parameters

struct gpio_chip * chip GPIO chip

Description

Free interrupts associated with GPIO ACPI event method for the given GPIO chip.

int **acpi_dev_gpio_irq_get**(struct acpi_device * *adev*, int *index*)
Find GpioInt and translate it to Linux IRQ number

Parameters

struct acpi_device * adev pointer to a ACPI device to get IRQ from

int index index of GpioInt resource (starting from `0`)

Description

If the device has one or more GpioInt resources, this function can be used to translate from the GPIO offset in the resource to the Linux IRQ number.

The function is idempotent, though each time it runs it will configure GPIO pin direction according to the flags in GpioInt resource.

Return

Linux IRQ number (> `0`) on success, negative errno on failure.

Device tree support

int **of_gpio_simple_xlate**(struct *gpio_chip* * *gc*, const struct of_phandle_args * *gpiospec*, u32 * *flags*)
translate gpiospec to the GPIO number and flags

Parameters

struct gpio_chip * gc pointer to the gpio_chip structure

`const struct of_phandle_args * gpiospec` GPIO specifier as found in the device tree

`u32 * flags` a flags pointer to fill in

Description

This is simple translation function, suitable for the most 1:1 mapped GPIO chips. This function performs only one sanity check: whether GPIO is less than ngpios (that is specified in the gpio_chip).

int **`of_mm_gpiochip_add_data`**(struct device_node * *np*, struct of_mm_gpio_chip * *mm_gc*, void * *data*)

Add memory mapped GPIO chip (bank)

Parameters

`struct device_node * np` device node of the GPIO chip

`struct of_mm_gpio_chip * mm_gc` pointer to the of_mm_gpio_chip allocated structure

`void * data` driver data to store in the struct gpio_chip

Description

To use this function you should allocate and fill mm_gc with:

1. In the gpio_chip structure: - all the callbacks - of_gpio_n_cells - of_xlate callback (optional)
3. In the of_mm_gpio_chip structure: - save_regs callback (optional)

If succeeded, this function will map bank's memory and will do all necessary work for you. Then you'll able to use .regs to manage GPIOs from the callbacks.

void **`of_mm_gpiochip_remove`**(struct of_mm_gpio_chip * *mm_gc*)

Remove memory mapped GPIO chip (bank)

Parameters

`struct of_mm_gpio_chip * mm_gc` pointer to the of_mm_gpio_chip allocated structure

Device-managed API

struct gpio_desc * **`devm_gpiod_get`**(struct *device* * *dev*, const char * *con_id*, enum gpiod_flags *flags*)

Resource-managed *`gpiod_get()`*

Parameters

`struct device * dev` GPIO consumer

`const char * con_id` function within the GPIO consumer

`enum gpiod_flags flags` optional GPIO initialization flags

Description

Managed *`gpiod_get()`*. GPIO descriptors returned from this function are automatically disposed on driver detach. See *`gpiod_get()`* for detailed information about behavior and return values.

struct gpio_desc * **`devm_gpiod_get_optional`**(struct *device* * *dev*, const char * *con_id*, enum gpiod_flags *flags*)

Resource-managed *`gpiod_get_optional()`*

Parameters

`struct device * dev` GPIO consumer

`const char * con_id` function within the GPIO consumer

`enum gpiod_flags flags` optional GPIO initialization flags

Description

Managed *gpiod_get_optional()*. GPIO descriptors returned from this function are automatically disposed on driver detach. See *gpiod_get_optional()* for detailed information about behavior and return values.

struct gpio_desc * **devm_gpiod_get_index**(struct *device* * *dev*, const char * *con_id*, unsigned int *idx*, enum gpiod_flags *flags*)

Resource-managed *gpiod_get_index()*

Parameters

struct device * dev GPIO consumer

const char * con_id function within the GPIO consumer

unsigned int idx index of the GPIO to obtain in the consumer

enum gpiod_flags flags optional GPIO initialization flags

Description

Managed *gpiod_get_index()*. GPIO descriptors returned from this function are automatically disposed on driver detach. See *gpiod_get_index()* for detailed information about behavior and return values.

struct gpio_desc * **devm_gpiod_get_from_of_node**(struct *device* * *dev*, struct device_node * *node*, const char * *propname*, int *index*, enum gpiod_flags *dflags*, const char * *label*)

obtain a GPIO from an OF node

Parameters

struct device * dev device for lifecycle management

struct device_node * node handle of the OF node

const char * propname name of the DT property representing the GPIO

int index index of the GPIO to obtain for the consumer

enum gpiod_flags dflags GPIO initialization flags

const char * label label to attach to the requested GPIO

Return

On successful request the GPIO pin is configured in accordance with provided **dflags**.

In case of error an ERR_PTR() is returned.

struct gpio_desc * **devm_fwnode_get_index_gpiod_from_child**(struct *device* * *dev*, const char * *con_id*, int *index*, struct fwnode_handle * *child*, enum gpiod_flags *flags*, const char * *label*)

get a GPIO descriptor from a device's child node

Parameters

struct device * dev GPIO consumer

const char * con_id function within the GPIO consumer

int index index of the GPIO to obtain in the consumer

struct fwnode_handle * child firmware node (child of **dev**)

enum gpiod_flags flags GPIO initialization flags

const char * label label to attach to the requested GPIO

Description

GPIO descriptors returned from this function are automatically disposed on driver detach.

On successful request the GPIO pin is configured in accordance with provided **flags**.

struct gpio_desc * **devm_gpiod_get_index_optional**(struct *device* * *dev*, const char * *con_id*, unsigned int *index*, enum gpiod_flags *flags*)

Resource-managed *gpiod_get_index_optional()*

Parameters

struct device * dev GPIO consumer

const char * con_id function within the GPIO consumer

unsigned int index index of the GPIO to obtain in the consumer

enum gpiod_flags flags optional GPIO initialization flags

Description

Managed *gpiod_get_index_optional()*. GPIO descriptors returned from this function are automatically disposed on driver detach. See *gpiod_get_index_optional()* for detailed information about behavior and return values.

struct gpio_descs * **devm_gpiod_get_array**(struct *device* * *dev*, const char * *con_id*, enum gpiod_flags *flags*)

Resource-managed *gpiod_get_array()*

Parameters

struct device * dev GPIO consumer

const char * con_id function within the GPIO consumer

enum gpiod_flags flags optional GPIO initialization flags

Description

Managed *gpiod_get_array()*. GPIO descriptors returned from this function are automatically disposed on driver detach. See *gpiod_get_array()* for detailed information about behavior and return values.

struct gpio_descs * **devm_gpiod_get_array_optional**(struct *device* * *dev*, const char * *con_id*, enum gpiod_flags *flags*)

Resource-managed *gpiod_get_array_optional()*

Parameters

struct device * dev GPIO consumer

const char * con_id function within the GPIO consumer

enum gpiod_flags flags optional GPIO initialization flags

Description

Managed *gpiod_get_array_optional()*. GPIO descriptors returned from this function are automatically disposed on driver detach. See *gpiod_get_array_optional()* for detailed information about behavior and return values.

void **devm_gpiod_put**(struct *device* * *dev*, struct gpio_desc * *desc*)

Resource-managed *gpiod_put()*

Parameters

struct device * dev GPIO consumer

struct gpio_desc * desc GPIO descriptor to dispose of

Description

Dispose of a GPIO descriptor obtained with *devm_gpiod_get()* or *devm_gpiod_get_index()*. Normally this function will not be called as the GPIO will be disposed of by the resource management code.

void **devm_gpiod_put_array**(struct *device* * *dev*, struct gpio_descs * *descs*)
Resource-managed *gpiod_put_array()*

Parameters

struct device * dev GPIO consumer

struct gpio_descs * descs GPIO descriptor array to dispose of

Description

Dispose of an array of GPIO descriptors obtained with *devm_gpiod_get_array()*. Normally this function will not be called as the GPIOs will be disposed of by the resource management code.

int **devm_gpio_request**(struct *device* * *dev*, unsigned *gpio*, const char * *label*)
request a GPIO for a managed device

Parameters

struct device * dev device to request the GPIO for

unsigned gpio GPIO to allocate

const char * label the name of the requested GPIO

Description

Except for the extra **dev** argument, this function takes the same arguments and performs the same function as `gpio_request()`. GPIOs requested with this function will be automatically freed on driver detach.

If an GPIO allocated with this function needs to be freed separately, *devm_gpio_free()* must be used.

int **devm_gpio_request_one**(struct *device* * *dev*, unsigned *gpio*, unsigned long *flags*, const char * *label*)
request a single GPIO with initial setup

Parameters

struct device * dev device to request for

unsigned gpio the GPIO number

unsigned long flags GPIO configuration as specified by GPIOF_*

const char * label a literal description string of this GPIO

void **devm_gpio_free**(struct *device* * *dev*, unsigned int *gpio*)
free a GPIO

Parameters

struct device * dev device to free GPIO for

unsigned int gpio GPIO to free

Description

Except for the extra **dev** argument, this function takes the same arguments and performs the same function as `gpio_free()`. This function instead of `gpio_free()` should be used to manually free GPIOs allocated with *devm_gpio_request()*.

sysfs helpers

int **gpiod_export**(struct gpio_desc * *desc*, bool *direction_may_change*)
    export a GPIO through sysfs

Parameters

struct gpio_desc * desc GPIO to make available, already requested

bool direction_may_change true if userspace may change GPIO direction

Context

arch_initcall or later

Description

When drivers want to make a GPIO accessible to userspace after they have requested it – perhaps while debugging, or as part of their public interface – they may use this routine. If the GPIO can change direction (some can't) and the caller allows it, userspace will see "direction" sysfs attribute which may be used to change the gpio's direction. A "value" attribute will always be provided.

Returns zero on success, else an error.

int **gpiod_export_link**(struct *device* * *dev*, const char * *name*, struct gpio_desc * *desc*)
    create a sysfs link to an exported GPIO node

Parameters

struct device * dev device under which to create symlink

const char * name name of the symlink

struct gpio_desc * desc GPIO to create symlink to, already exported

Description

Set up a symlink from /sys/.../dev/name to /sys/class/gpio/gpioN node. Caller is responsible for unlinking.

Returns zero on success, else an error.

void **gpiod_unexport**(struct gpio_desc * *desc*)
    reverse effect of *gpiod_export()*

Parameters

struct gpio_desc * desc GPIO to make unavailable

Description

This is implicit on `gpiod_free()`.

CHAPTER

THIRTYFIVE

MISCELLANEOUS DEVICES

int **misc_register**(struct miscdevice * *misc*)
  register a miscellaneous device

Parameters

struct miscdevice * misc device structure

Description

Register a miscellaneous device with the kernel. If the minor number is set to `MISC_DYNAMIC_MINOR` a minor number is assigned and placed in the minor field of the structure. For other cases the minor number requested is used.

The structure passed is linked into the kernel and may not be destroyed until it has been unregistered. By default, an `open()` syscall to the device sets file->private_data to point to the structure. Drivers don't need open in fops for this.

A zero is returned on success and a negative errno code for failure.

void **misc_deregister**(struct miscdevice * *misc*)
  unregister a miscellaneous device

Parameters

struct miscdevice * misc device to unregister

Description

Unregister a miscellaneous device that was previously successfully registered with *misc_register()*.

CHAPTER

THIRTYSIX

DMAENGINE DOCUMENTATION

DMAEngine documentation provides documents for various aspects of DMAEngine framework.

DMAEngine documentation

This book helps with DMAengine internal APIs and guide for DMAEngine device driver writers.

DMAengine controller documentation

Hardware Introduction

Most of the Slave DMA controllers have the same general principles of operations.

They have a given number of channels to use for the DMA transfers, and a given number of requests lines.

Requests and channels are pretty much orthogonal. Channels can be used to serve several to any requests. To simplify, channels are the entities that will be doing the copy, and requests what endpoints are involved.

The request lines actually correspond to physical lines going from the DMA-eligible devices to the controller itself. Whenever the device will want to start a transfer, it will assert a DMA request (DRQ) by asserting that request line.

A very simple DMA controller would only take into account a single parameter: the transfer size. At each clock cycle, it would transfer a byte of data from one buffer to another, until the transfer size has been reached.

That wouldn't work well in the real world, since slave devices might require a specific number of bits to be transferred in a single cycle. For example, we may want to transfer as much data as the physical bus allows to maximize performances when doing a simple memory copy operation, but our audio device could have a narrower FIFO that requires data to be written exactly 16 or 24 bits at a time. This is why most if not all of the DMA controllers can adjust this, using a parameter called the transfer width.

Moreover, some DMA controllers, whenever the RAM is used as a source or destination, can group the reads or writes in memory into a buffer, so instead of having a lot of small memory accesses, which is not really efficient, you'll get several bigger transfers. This is done using a parameter called the burst size, that defines how many single reads/writes it's allowed to do without the controller splitting the transfer into smaller sub-transfers.

Our theoretical DMA controller would then only be able to do transfers that involve a single contiguous block of data. However, some of the transfers we usually have are not, and want to copy data from non-contiguous buffers to a contiguous buffer, which is called scatter-gather.

DMAEngine, at least for mem2dev transfers, require support for scatter-gather. So we're left with two cases here: either we have a quite simple DMA controller that doesn't support it, and we'll have to implement it in software, or we have a more advanced DMA controller, that implements in hardware scatter-gather.

The latter are usually programmed using a collection of chunks to transfer, and whenever the transfer is started, the controller will go over that collection, doing whatever we programmed there.

This collection is usually either a table or a linked list. You will then push either the address of the table and its number of elements, or the first item of the list to one channel of the DMA controller, and whenever a DRQ will be asserted, it will go through the collection to know where to fetch the data from.

Either way, the format of this collection is completely dependent on your hardware. Each DMA controller will require a different structure, but all of them will require, for every chunk, at least the source and destination addresses, whether it should increment these addresses or not and the three parameters we saw earlier: the burst size, the transfer width and the transfer size.

The one last thing is that usually, slave devices won't issue DRQ by default, and you have to enable this in your slave device driver first whenever you're willing to use DMA.

These were just the general memory-to-memory (also called mem2mem) or memory-to-device (mem2dev) kind of transfers. Most devices often support other kind of transfers or memory operations that dmaengine support and will be detailed later in this document.

DMA Support in Linux

Historically, DMA controller drivers have been implemented using the async TX API, to offload operations such as memory copy, XOR, cryptography, etc., basically any memory to memory operation.

Over time, the need for memory to device transfers arose, and dmaengine was extended. Nowadays, the async TX API is written as a layer on top of dmaengine, and acts as a client. Still, dmaengine accommodates that API in some cases, and made some design choices to ensure that it stayed compatible.

For more information on the Async TX API, please look the relevant documentation file in Documentation/crypto/async-tx-api.txt.

DMAEngine APIs

`struct dma_device` Initialization

Just like any other kernel framework, the whole DMAEngine registration relies on the driver filling a structure and registering against the framework. In our case, that structure is dma_device.

The first thing you need to do in your driver is to allocate this structure. Any of the usual memory allocators will do, but you'll also need to initialize a few fields in there:

- `channels`: should be initialized as a list using the INIT_LIST_HEAD macro for example
- `src_addr_widths`: should contain a bitmask of the supported source transfer width
- `dst_addr_widths`: should contain a bitmask of the supported destination transfer width
- `directions`: should contain a bitmask of the supported slave directions (i.e. excluding mem2mem transfers)
- `residue_granularity`: granularity of the transfer residue reported to dma_set_residue. This can be either:
  - Descriptor: your device doesn't support any kind of residue reporting. The framework will only know that a particular transaction descriptor is done.
  - Segment: your device is able to report which chunks have been transferred
  - Burst: your device is able to report which burst have been transferred
- dev: should hold the pointer to the `struct device` associated to your current driver instance.

Supported transaction types

The next thing you need is to set which transaction types your device (and driver) supports.

Our `dma_device structure` has a field called cap_mask that holds the various types of transaction supported, and you need to modify this mask using the dma_cap_set function, with various flags depending on transaction types you support as an argument.

All those capabilities are defined in the `dma_transaction_type enum`, in `include/linux/dmaengine.h`

Currently, the types available are:

- DMA_MEMCPY
  - The device is able to do memory to memory copies
- DMA_XOR
  - The device is able to perform XOR operations on memory areas
  - Used to accelerate XOR intensive tasks, such as RAID5
- DMA_XOR_VAL
  - The device is able to perform parity check using the XOR algorithm against a memory buffer.
- DMA_PQ
  - The device is able to perform RAID6 P+Q computations, P being a simple XOR, and Q being a Reed-Solomon algorithm.
- DMA_PQ_VAL
  - The device is able to perform parity check using RAID6 P+Q algorithm against a memory buffer.
- DMA_INTERRUPT
  - The device is able to trigger a dummy transfer that will generate periodic interrupts
  - Used by the client drivers to register a callback that will be called on a regular basis through the DMA controller interrupt
- DMA_PRIVATE
  - The devices only supports slave transfers, and as such isn’t available for async transfers.
- DMA_ASYNC_TX
  - Must not be set by the device, and will be set by the framework if needed
  - TODO: What is it about?
- DMA_SLAVE
  - The device can handle device to memory transfers, including scatter-gather transfers.
  - While in the mem2mem case we were having two distinct types to deal with a single chunk to copy or a collection of them, here, we just have a single transaction type that is supposed to handle both.
  - If you want to transfer a single contiguous memory buffer, simply build a scatter list with only one item.
- DMA_CYCLIC
  - The device can handle cyclic transfers.
  - A cyclic transfer is a transfer where the chunk collection will loop over itself, with the last item pointing to the first.
  - It’s usually used for audio transfers, where you want to operate on a single ring buffer that you will fill with your audio data.

- DMA_INTERLEAVE
  - The device supports interleaved transfer.
  - These transfers can transfer data from a non-contiguous buffer to a non-contiguous buffer, opposed to DMA_SLAVE that can transfer data from a non-contiguous data set to a continuous destination buffer.
  - It's usually used for 2d content transfers, in which case you want to transfer a portion of uncompressed data directly to the display to print it

These various types will also affect how the source and destination addresses change over time.

Addresses pointing to RAM are typically incremented (or decremented) after each transfer. In case of a ring buffer, they may loop (DMA_CYCLIC). Addresses pointing to a device's register (e.g. a FIFO) are typically fixed.

Device operations

Our dma_device structure also requires a few function pointers in order to implement the actual logic, now that we described what operations we were able to perform.

The functions that we have to fill in there, and hence have to implement, obviously depend on the transaction types you reported as supported.

- `device_alloc_chan_resources`
- `device_free_chan_resources`
  - These functions will be called whenever a driver will call `dma_request_channel` or `dma_release_channel` for the first/last time on the channel associated to that driver.
  - They are in charge of allocating/freeing all the needed resources in order for that channel to be useful for your driver.
  - These functions can sleep.
- `device_prep_dma_*`
  - These functions are matching the capabilities you registered previously.
  - These functions all take the buffer or the scatterlist relevant for the transfer being prepared, and should create a hardware descriptor or a list of hardware descriptors from it
  - These functions can be called from an interrupt context
  - Any allocation you might do should be using the GFP_NOWAIT flag, in order not to potentially sleep, but without depleting the emergency pool either.
  - Drivers should try to pre-allocate any memory they might need during the transfer setup at probe time to avoid putting to much pressure on the nowait allocator.
  - It should return a unique instance of the `dma_async_tx_descriptor structure`, that further represents this particular transfer.
  - This structure can be initialized using the function `dma_async_tx_descriptor_init`.
  - You'll also need to set two fields in this structure:
    * flags: TODO: Can it be modified by the driver itself, or should it be always the flags passed in the arguments
    * tx_submit: A pointer to a function you have to implement, that is supposed to push the current transaction descriptor to a pending queue, waiting for issue_pending to be called.
  - In this structure the function pointer callback_result can be initialized in order for the submitter to be notified that a transaction has completed. In the earlier code the function pointer callback has been used. However it does not provide any status to the transaction and will be deprecated.

The result structure defined as `dmaengine_result` that is passed in to callback_result has two fields:

  * result: This provides the transfer result defined by `dmaengine_tx_result`. Either success or some error condition.
  * residue: Provides the residue bytes of the transfer for those that support residue.

- `device_issue_pending`
  - Takes the first transaction descriptor in the pending queue, and starts the transfer. Whenever that transfer is done, it should move to the next transaction in the list.
  - This function can be called in an interrupt context
- `device_tx_status`
  - Should report the bytes left to go over on the given channel
  - Should only care about the transaction descriptor passed as argument, not the currently active one on a given channel
  - The tx_state argument might be NULL
  - Should use dma_set_residue to report it
  - In the case of a cyclic transfer, it should only take into account the current period.
  - This function can be called in an interrupt context.
- device_config
  - Reconfigures the channel with the configuration given as argument
  - This command should NOT perform synchronously, or on any currently queued transfers, but only on subsequent ones
  - In this case, the function will receive a `dma_slave_config` structure pointer as an argument, that will detail which configuration to use.
  - Even though that structure contains a direction field, this field is deprecated in favor of the direction argument given to the prep_* functions
  - This call is mandatory for slave operations only. This should NOT be set or expected to be set for memcpy operations. If a driver support both, it should use this call for slave operations only and not for memcpy ones.
- device_pause
  - Pauses a transfer on the channel
  - This command should operate synchronously on the channel, pausing right away the work of the given channel
- device_resume
  - Resumes a transfer on the channel
  - This command should operate synchronously on the channel, resuming right away the work of the given channel
- device_terminate_all
  - Aborts all the pending and ongoing transfers on the channel
  - For aborted transfers the complete callback should not be called
  - Can be called from atomic context or from within a complete callback of a descriptor. Must not sleep. Drivers must be able to handle this correctly.
  - Termination may be asynchronous. The driver does not have to wait until the currently active transfer has completely stopped. See device_synchronize.

- device_synchronize
  - Must synchronize the termination of a channel to the current context.
  - Must make sure that memory for previously submitted descriptors is no longer accessed by the DMA controller.
  - Must make sure that all complete callbacks for previously submitted descriptors have finished running and none are scheduled to run.
  - May sleep.

Misc notes

(stuff that should be documented, but don't really know where to put them)

`dma_run_dependencies`

- Should be called at the end of an async TX transfer, and can be ignored in the slave transfers case.
- Makes sure that dependent operations are run before marking it as complete.

dma_cookie_t

- it's a DMA transaction ID that will increment over time.
- Not really relevant any more since the introduction of `virt-dma` that abstracts it away.

DMA_CTRL_ACK

- If clear, the descriptor cannot be reused by provider until the client acknowledges receipt, i.e. has has a chance to establish any dependency chains
- This can be acked by invoking async_tx_ack()
- If set, does not mean descriptor can be reused

DMA_CTRL_REUSE

- If set, the descriptor can be reused after being completed. It should not be freed by provider if this flag is set.
- The descriptor should be prepared for reuse by invoking `dmaengine_desc_set_reuse()` which will set DMA_CTRL_REUSE.
- `dmaengine_desc_set_reuse()` will succeed only when channel support reusable descriptor as exhibited by capabilities
- As a consequence, if a device driver wants to skip the `dma_map_sg()` and `dma_unmap_sg()` in between 2 transfers, because the DMA'd data wasn't used, it can resubmit the transfer right after its completion.
- Descriptor can be freed in few ways
  - Clearing DMA_CTRL_REUSE by invoking `dmaengine_desc_clear_reuse()` and submitting for last txn
  - Explicitly invoking `dmaengine_desc_free()`, this can succeed only when DMA_CTRL_REUSE is already set
  - Terminating the channel
- DMA_PREP_CMD
  - If set, the client driver tells DMA controller that passed data in DMA API is command data.
  - Interpretation of command data is DMA controller specific. It can be used for issuing commands to other peripherals/register reads/register writes for which the descriptor should be in different format from normal data descriptors.

General Design Notes

Most of the DMAEngine drivers you'll see are based on a similar design that handles the end of transfer interrupts in the handler, but defer most work to a tasklet, including the start of a new transfer whenever the previous transfer ended.

This is a rather inefficient design though, because the inter-transfer latency will be not only the interrupt latency, but also the scheduling latency of the tasklet, which will leave the channel idle in between, which will slow down the global transfer rate.

You should avoid this kind of practice, and instead of electing a new transfer in your tasklet, move that part to the interrupt handler in order to have a shorter idle window (that we can't really avoid anyway).

Glossary

- Burst: A number of consecutive read or write operations that can be queued to buffers before being flushed to memory.
- Chunk: A contiguous collection of bursts
- Transfer: A collection of chunks (be it contiguous or not)

DMAEngine client documentation

This book is a guide to device driver writers on how to use the Slave-DMA API of the DMAEngine. This is applicable only for slave DMA usage only.

DMA Engine API Guide

Vinod Koul <vinod dot koul at intel.com>

Note:

For DMA Engine usage in async_tx please see: `Documentation/crypto/async-tx-api.txt`

Below is a guide to device driver writers on how to use the Slave-DMA API of the DMA Engine. This is applicable only for slave DMA usage only.

DMA usage

The slave DMA usage consists of following steps:

- Allocate a DMA slave channel
- Set slave and controller specific parameters
- Get a descriptor for transaction
- Submit the transaction
- Issue pending requests and wait for callback notification

The details of these operations are:

1. Allocate a DMA slave channel

   Channel allocation is slightly different in the slave DMA context, client drivers typically need a channel from a particular DMA controller only and even in some cases a specific channel is desired. To request a channel dma_request_chan() API is used.

Interface:

```
struct dma_chan *dma_request_chan(struct device *dev, const char *name);
```

Which will find and return the name DMA channel associated with the 'dev' device. The association is done via DT, ACPI or board file based dma_slave_map matching table.

A channel allocated via this interface is exclusive to the caller, until dma_release_channel() is called.

2. Set slave and controller specific parameters

   Next step is always to pass some specific information to the DMA driver. Most of the generic information which a slave DMA can use is in struct dma_slave_config. This allows the clients to specify DMA direction, DMA addresses, bus widths, DMA burst lengths etc for the peripheral.

   If some DMA controllers have more parameters to be sent then they should try to embed struct dma_slave_config in their controller specific structure. That gives flexibility to client to pass more parameters, if required.

   Interface:

```
int dmaengine_slave_config(struct dma_chan *chan,
                  struct dma_slave_config *config)
```

   Please see the dma_slave_config structure definition in dmaengine.h for a detailed explanation of the struct members. Please note that the 'direction' member will be going away as it duplicates the direction given in the prepare call.

3. Get a descriptor for transaction

   For slave usage the various modes of slave transfers supported by the DMA-engine are:

   - slave_sg: DMA a list of scatter gather buffers from/to a peripheral
   - dma_cyclic: Perform a cyclic DMA operation from/to a peripheral till the operation is explicitly stopped.
   - interleaved_dma: This is common to Slave as well as M2M clients. For slave address of devices' fifo could be already known to the driver. Various types of operations could be expressed by setting appropriate values to the 'dma_interleaved_template' members.

   A non-NULL return of this transfer API represents a "descriptor" for the given transaction.

   Interface:

```
struct dma_async_tx_descriptor *dmaengine_prep_slave_sg(
          struct dma_chan *chan, struct scatterlist *sgl,
          unsigned int sg_len, enum dma_data_direction direction,
          unsigned long flags);

struct dma_async_tx_descriptor *dmaengine_prep_dma_cyclic(
          struct dma_chan *chan, dma_addr_t buf_addr, size_t buf_len,
          size_t period_len, enum dma_data_direction direction);

struct dma_async_tx_descriptor *dmaengine_prep_interleaved_dma(
          struct dma_chan *chan, struct dma_interleaved_template *xt,
          unsigned long flags);
```

   The peripheral driver is expected to have mapped the scatterlist for the DMA operation prior to calling dmaengine_prep_slave_sg(), and must keep the scatterlist mapped until the DMA operation has completed. The scatterlist must be mapped using the DMA struct device. If a mapping needs to be synchronized later, dma_sync_*_for_*() must be called using the DMA struct device, too. So, normal setup should look like this:

```
nr_sg = dma_map_sg(chan->device->dev, sgl, sg_len);
   if (nr_sg == 0)
           /* error */
   desc = dmaengine_prep_slave_sg(chan, sgl, nr_sg, direction, flags);
```

Once a descriptor has been obtained, the callback information can be added and the descriptor must then be submitted. Some DMA engine drivers may hold a spinlock between a successful preparation and submission so it is important that these two operations are closely paired.

> ***Note:***
>
> *Although the async_tx API specifies that completion callback routines cannot submit any new operations, this is not the case for slave/cyclic DMA.*
> *For slave DMA, the subsequent transaction may not be available for submission prior to callback function being invoked, so slave DMA callbacks are permitted to prepare and submit a new transaction.*
> *For cyclic DMA, a callback function may wish to terminate the DMA via dmaengine_terminate_async().*
> *Therefore, it is important that DMA engine drivers drop any locks before calling the callback function which may cause a deadlock.*
> *Note that callbacks will always be invoked from the DMA engines tasklet, never from interrupt context.*

4. Submit the transaction

   Once the descriptor has been prepared and the callback information added, it must be placed on the DMA engine drivers pending queue.

   Interface:

```
dma_cookie_t dmaengine_submit(struct dma_async_tx_descriptor *desc)
```

   This returns a cookie can be used to check the progress of DMA engine activity via other DMA engine calls not covered in this document.

   dmaengine_submit() will not start the DMA operation, it merely adds it to the pending queue. For this, see step 5, dma_async_issue_pending.

5. Issue pending DMA requests and wait for callback notification

   The transactions in the pending queue can be activated by calling the issue_pending API. If channel is idle then the first transaction in queue is started and subsequent ones queued up.

   On completion of each DMA operation, the next in queue is started and a tasklet triggered. The tasklet will then call the client driver completion callback routine for notification, if set.

   Interface:

```
void dma_async_issue_pending(struct dma_chan *chan);
```

Further APIs:

1. Terminate APIs

```
int dmaengine_terminate_sync(struct dma_chan *chan)
int dmaengine_terminate_async(struct dma_chan *chan)
int dmaengine_terminate_all(struct dma_chan *chan) /* DEPRECATED */
```

   This causes all activity for the DMA channel to be stopped, and may discard data in the DMA FIFO which hasn't been fully transferred. No callback functions will be called for any incomplete transfers.

   Two variants of this function are available.

dmaengine_terminate_async() might not wait until the DMA has been fully stopped or until any running complete callbacks have finished. But it is possible to call dmaengine_terminate_async() from atomic context or from within a complete callback. dmaengine_synchronize() must be called before it is safe to free the memory accessed by the DMA transfer or free resources accessed from within the complete callback.

dmaengine_terminate_sync() will wait for the transfer and any running complete callbacks to finish before it returns. But the function must not be called from atomic context or from within a complete callback.

dmaengine_terminate_all() is deprecated and should not be used in new code.

2. Pause API

```
int dmaengine_pause(struct dma_chan *chan)
```

This pauses activity on the DMA channel without data loss.

3. Resume API

```
int dmaengine_resume(struct dma_chan *chan)
```

Resume a previously paused DMA channel. It is invalid to resume a channel which is not currently paused.

4. Check Txn complete

```
enum dma_status dma_async_is_tx_complete(struct dma_chan *chan,
          dma_cookie_t cookie, dma_cookie_t *last, dma_cookie_t *used)
```

This can be used to check the status of the channel. Please see the documentation in include/linux/dmaengine.h for a more complete description of this API.

This can be used in conjunction with dma_async_is_complete() and the cookie returned from dmaengine_submit() to check for completion of a specific DMA transaction.

Note:

Not all DMA engine drivers can return reliable information for a running DMA channel. It is recommended that DMA engine users pause or stop (via dmaengine_terminate_all()) the channel before using this API.

5. Synchronize termination API

```
void dmaengine_synchronize(struct dma_chan *chan)
```

Synchronize the termination of the DMA channel to the current context.

This function should be used after dmaengine_terminate_async() to synchronize the termination of the DMA channel to the current context. The function will wait for the transfer and any running complete callbacks to finish before it returns.

If dmaengine_terminate_async() is used to stop the DMA channel this function must be called before it is safe to free memory accessed by previously submitted descriptors or to free any resources accessed within the complete callback of previously submitted descriptors.

The behavior of this function is undefined if dma_async_issue_pending() has been called between dmaengine_terminate_async() and this function.

DMA Test documentation

This book introduces how to test DMA drivers using dmatest module.

DMA Test Guide

Andy Shevchenko <andriy.shevchenko@linux.intel.com>

This small document introduces how to test DMA drivers using dmatest module.

Part 1 - How to build the test module

The menuconfig contains an option that could be found by following path: Device Drivers -> DMA Engine support -> DMA Test client

In the configuration file the option called CONFIG_DMATEST. The dmatest could be built as module or inside kernel. Let's consider those cases.

Part 2 - When dmatest is built as a module

Example of usage:

```
% modprobe dmatest channel=dma0chan0 timeout=2000 iterations=1 run=1
```

…or:

```
% modprobe dmatest
% echo dma0chan0 > /sys/module/dmatest/parameters/channel
% echo 2000 > /sys/module/dmatest/parameters/timeout
% echo 1 > /sys/module/dmatest/parameters/iterations
% echo 1 > /sys/module/dmatest/parameters/run
```

…or on the kernel command line:

```
dmatest.channel=dma0chan0 dmatest.timeout=2000 dmatest.iterations=1 dmatest.run=1
```

..hint:: available channel list could be extracted by running the following command:

```
% ls -1 /sys/class/dma/
```

Once started a message like "dmatest: Started 1 threads using dma0chan0" is emitted. After that only test failure messages are reported until the test stops.

Note that running a new test will not stop any in progress test.

The following command returns the state of the test.

```
% cat /sys/module/dmatest/parameters/run
```

To wait for test completion userpace can poll 'run' until it is false, or use the wait parameter. Specifying 'wait=1' when loading the module causes module initialization to pause until a test run has completed, while reading /sys/module/dmatest/parameters/wait waits for any running test to complete before returning. For example, the following scripts wait for 42 tests to complete before exiting. Note that if 'iterations' is set to 'infinite' then waiting is disabled.

Example:

```
% modprobe dmatest run=1 iterations=42 wait=1
% modprobe -r dmatest
```

…or:

```
% modprobe dmatest run=1 iterations=42
% cat /sys/module/dmatest/parameters/wait
% modprobe -r dmatest
```

Part 3 - When built-in in the kernel

The module parameters that is supplied to the kernel command line will be used for the first performed test. After user gets a control, the test could be re-run with the same or different parameters. For the details see the above section "Part 2 - When dmatest is built as a module…"

In both cases the module parameters are used as the actual values for the test case. You always could check them at run-time by running

```
% grep -H . /sys/module/dmatest/parameters/*
```

Part 4 - Gathering the test results

Test results are printed to the kernel log buffer with the format:

```
"dmatest: result <channel>: <test id>: '<error msg>' with src_off=<val> dst_off=<val> len=<val> (<err co
```

Example of output:

```
% dmesg | tail -n 1
dmatest: result dma0chan0-copy0: #1: No errors with src_off=0x7bf dst_off=0x8ad len=0x3fea (0)
```

The message format is unified across the different types of errors. A number in the parens represents additional information, e.g. error code, error counter, or status. A test thread also emits a summary line at completion listing the number of tests executed, number that failed, and a result code.

Example:

```
% dmesg | tail -n 1
dmatest: dma0chan0-copy0: summary 1 test, 0 failures 1000 iops 100000 KB/s (0)
```

The details of a data miscompare error are also emitted, but do not follow the above format.

PXA DMA documentation

This book adds some notes about PXA DMA

PXA/MMP - DMA Slave controller

Constraints

a) Transfers hot queuing A driver submitting a transfer and issuing it should be granted the transfer is queued even on a running DMA channel. This implies that the queuing doesn't wait for the previous transfer end, and that the descriptor chaining is not only done in the irq/tasklet code triggered by the end of the transfer. A transfer which is submitted and issued on a phy doesn't wait for a phy to stop and restart, but is submitted on a "running channel". The other drivers, especially mmp_pdma waited for the phy to stop before relaunching a new transfer.

b) All transfers having asked for confirmation should be signaled Any issued transfer with DMA_PREP_INTERRUPT should trigger a callback call. This implies that even if an irq/tasklet is triggered by end of tx1, but at the time of irq/dma tx2 is already finished, tx1->complete() and tx2->complete() should be called.

c) Channel running state A driver should be able to query if a channel is running or not. For the multimedia case, such as video capture, if a transfer is submitted and then a check of the DMA channel reports a "stopped channel", the transfer should not be issued until the next "start of frame interrupt", hence the need to know if a channel is in running or stopped state.

d) Bandwidth guarantee The PXA architecture has 4 levels of DMAs priorities : high, normal, low. The high priorities get twice as much bandwidth as the normal, which get twice as much as the low priorities. A driver should be able to request a priority, especially the real-time ones such as pxa_camera with (big) throughputs.

Design

a) Virtual channels Same concept as in sa11x0 driver, ie. a driver was assigned a "virtual channel" linked to the requestor line, and the physical DMA channel is assigned on the fly when the transfer is issued.

2. Transfer anatomy for a scatter-gather transfer

```
+------------+-----+---------------+----------------+-----------------+
| desc-sg[0] | ... | desc-sg[last] | status updater | finisher/linker |
+------------+-----+---------------+----------------+-----------------+
```

This structure is pointed by dma->sg_cpu. The descriptors are used as follows :

- desc-sg[i]: i-th descriptor, transferring the i-th sg element to the video buffer scatter gather
- status updater Transfers a single u32 to a well known dma coherent memory to leave a trace that this transfer is done. The "well known" is unique per physical channel, meaning that a read of this value will tell which is the last finished transfer at that point in time.
- finisher: has ddadr=DADDR_STOP, dcmd=ENDIRQEN
- linker: has ddadr= desc-sg[0] of next transfer, dcmd=0

c) Transfers hot-chaining Suppose the running chain is:

```
Buffer 1              Buffer 2
+---------+----+---+  +----+----+----+---+
| d0 | .. | dN | l |  | d0 | .. | dN | f |
+---------+----+-|-+  ^----+----+----+---+
                 |    |
                 +----+
```

After a call to dmaengine_submit(b3), the chain will look like:

```
Buffer 1              Buffer 2              Buffer 3
+---------+----+---+  +----+----+----+---+  +----+----+----+---+
| d0 | .. | dN | l |  | d0 | .. | dN | l |  | d0 | .. | dN | f |
+---------+----+-|-+  ^----+----+----+-|-+  ^----+----+----+---+
                 |    |                |    |
                 +----+                +----+
                                      new_link
```

If while new_link was created the DMA channel stopped, it is _not_ restarted. Hot-chaining doesn't break the assumption that dma_async_issue_pending() is to be used to ensure the transfer is actually started.

One exception to this rule :

- if Buffer1 and Buffer2 had all their addresses 8 bytes aligned
- and if Buffer3 has at least one address not 4 bytes aligned
- then hot-chaining cannot happen, as the channel must be stopped, the "align bit" must be set, and the channel restarted As a consequence, such a transfer tx_submit() will be queued on the submitted queue, and this specific case if the DMA is already running in aligned mode.

d) Transfers completion updater Each time a transfer is completed on a channel, an interrupt might be generated or not, up to the client's request. But in each case, the last descriptor of a transfer, the "status updater", will write the latest transfer being completed into the physical channel's completion mark.

This will speed up residue calculation, for large transfers such as video buffers which hold around 6k descriptors or more. This also allows without any lock to find out what is the latest completed transfer in a running DMA chain.

e) Transfers completion, irq and tasklet When a transfer flagged as "DMA_PREP_INTERRUPT" is finished, the dma irq is raised. Upon this interrupt, a tasklet is scheduled for the physical channel.

The tasklet is responsible for :

- reading the physical channel last updater mark
- calling all the transfer callbacks of finished transfers, based on that mark, and each transfer flags.

If a transfer is completed while this handling is done, a dma irq will be raised, and the tasklet will be scheduled once again, having a new updater mark.

f) Residue Residue granularity will be descriptor based. The issued but not completed transfers will be scanned for all of their descriptors against the currently running descriptor.

g) Most complicated case of driver's tx queues The most tricky situation is when :

  - there are not "acked" transfers (tx0)
  - a driver submitted an aligned tx1, not chained
  - a driver submitted an aligned tx2 => tx2 is cold chained to tx1
  - a driver issued tx1+tx2 => channel is running in aligned mode
  - a driver submitted an aligned tx3 => tx3 is hot-chained
  - a driver submitted an unaligned tx4 => tx4 is put in submitted queue, not chained
  - a driver issued tx4 => tx4 is put in issued queue, not chained
  - a driver submitted an aligned tx5 => tx5 is put in submitted queue, not chained
  - a driver submitted an aligned tx6 => tx6 is put in submitted queue, cold chained to tx5

  This translates into (after tx4 is issued) :

  - issued queue

```
    +-----+ +-----+ +-----+ +-----+
    | tx1 | | tx2 | | tx3 | | tx4 |
    +---|-+ ^---|-+ ^-----+ +-----+
        |   |   |   |
        +---+   +---+
      - submitted queue
    +-----+ +-----+
    | tx5 | | tx6 |
    +---|-+ ^-----+
        |   |
        +---+
```

- completed queue : empty
- allocated queue : tx0

It should be noted that after tx3 is completed, the channel is stopped, and restarted in "unaligned mode" to handle tx4.

Author: Robert Jarzmik <robert.jarzmik@free.fr>

CHAPTER

THIRTYSEVEN

LINUX KERNEL SLIMBUS SUPPORT

Overview

What is SLIMbus?

SLIMbus (Serial Low Power Interchip Media Bus) is a specification developed by MIPI (Mobile Industry Processor Interface) alliance. The bus uses master/slave configuration, and is a 2-wire multi-drop implementation (clock, and data).

Currently, SLIMbus is used to interface between application processors of SoCs (System-on-Chip) and peripheral components (typically codec). SLIMbus uses Time-Division-Multiplexing to accommodate multiple data channels, and a control channel.

The control channel is used for various control functions such as bus management, configuration and status updates. These messages can be unicast (e.g. reading/writing device specific values), or multicast (e.g. data channel reconfiguration sequence is a broadcast message announced to all devices)

A data channel is used for data-transfer between 2 SLIMbus devices. Data channel uses dedicated ports on the device.

Hardware description:

SLIMbus specification has different types of device classifications based on their capabilities. A manager device is responsible for enumeration, configuration, and dynamic channel allocation. Every bus has 1 active manager.

A generic device is a device providing application functionality (e.g. codec).

Framer device is responsible for clocking the bus, and transmitting frame-sync and framing information on the bus.

Each SLIMbus component has an interface device for monitoring physical layer.

Typically each SoC contains SLIMbus component having 1 manager, 1 framer device, 1 generic device (for data channel support), and 1 interface device. External peripheral SLIMbus component usually has 1 generic device (for functionality/data channel support), and an associated interface device. The generic device's registers are mapped as 'value elements' so that they can be written/read using SLIMbus control channel exchanging control/status type of information. In case there are multiple framer devices on the same bus, manager device is responsible to select the active-framer for clocking the bus.

Per specification, SLIMbus uses "clock gears" to do power management based on current frequency and bandwidth requirements. There are 10 clock gears and each gear changes the SLIMbus frequency to be twice its previous gear.

Each device has a 6-byte enumeration-address and the manager assigns every device with a 1-byte logical address after the devices report presence on the bus.

Software description:

There are 2 types of SLIMbus drivers:

slim_controller represents a 'controller' for SLIMbus. This driver should implement duties needed by the SoC (manager device, associated interface device for monitoring the layers and reporting errors, default framer device).

slim_device represents the 'generic device/component' for SLIMbus, and a slim_driver should implement driver for that slim_device.

Device notifications to the driver:

Since SLIMbus devices have mechanisms for reporting their presence, the framework allows drivers to bind when corresponding devices report their presence on the bus. However, it is possible that the driver needs to be probed first so that it can enable corresponding SLIMbus device (e.g. power it up and/or take it out of reset). To support that behavior, the framework allows drivers to probe first as well (e.g. using standard DeviceTree compatibility field). This creates the necessity for the driver to know when the device is functional (i.e. reported present). device_up callback is used for that reason when the device reports present and is assigned a logical address by the controller.

Similarly, SLIMbus devices 'report absent' when they go down. A 'device_down' callback notifies the driver when the device reports absent and its logical address assignment is invalidated by the controller.

Another notification "boot_device" is used to notify the slim_driver when controller resets the bus. This notification allows the driver to take necessary steps to boot the device so that it's functional after the bus has been reset.

Driver and Controller APIs:

struct **slim_eaddr**
Enumeration address for a SLIMbus device

Definition

```
struct slim_eaddr {
  u16 manf_id;
  u16 prod_code;
  u8 dev_index;
  u8 instance;
};
```

Members

manf_id Manufacturer Id for the device

prod_code Product code

dev_index Device index

instance Instance value

enum **slim_device_status**
slim device status

Constants

SLIM_DEVICE_STATUS_DOWN Slim device is absent or not reported yet.

SLIM_DEVICE_STATUS_UP Slim device is announced on the bus.

SLIM_DEVICE_STATUS_RESERVED Reserved for future use.

struct **slim_device**
Slim device handle.

Definition

```
struct slim_device {
  struct device           dev;
  struct slim_eaddr       e_addr;
  struct slim_controller  *ctrl;
  enum slim_device_status status;
  u8 laddr;
  bool is_laddr_valid;
};
```

Members

dev Driver model representation of the device.

e_addr Enumeration address of this device.

ctrl slim controller instance.

status slim device status

laddr 1-byte Logical address of this device.

is_laddr_valid indicates if the laddr is valid or not

Description

This is the client/device handle returned when a SLIMbus device is registered with a controller. Pointer to this structure is used by client-driver as a handle.

struct **slim_driver**
SLIMbus 'generic device' (slave) device driver (similar to 'spi_device' on SPI)

Definition

```
struct slim_driver {
  int (*probe)(struct slim_device *sl);
  void (*remove)(struct slim_device *sl);
  void (*shutdown)(struct slim_device *sl);
  int (*device_status)(struct slim_device *sl, enum slim_device_status s);
  struct device_driver            driver;
  const struct slim_device_id     *id_table;
};
```

Members

probe Binds this driver to a SLIMbus device.

remove Unbinds this driver from the SLIMbus device.

shutdown Standard shutdown callback used during powerdown/halt.

device_status This callback is called when - The device reports present and gets a laddr assigned - The device reports absent, or the bus goes down.

driver SLIMbus device drivers should initialize name and owner field of this structure

id_table List of SLIMbus devices supported by this driver

struct **slim_val_inf**
Slimbus value or information element

Definition

```
struct slim_val_inf {
  u16 start_offset;
  u8 num_bytes;
  u8 *rbuf;
  const u8                *wbuf;
  struct completion      *comp;
};
```

Members

`start_offset` Specifies starting offset in information/value element map

`num_bytes` upto 16. This ensures that the message will fit the slicesize per SLIMbus spec

`rbuf` buffer to read the values

`wbuf` buffer to write

`comp` completion for asynchronous operations, valid only if TID is required for transaction, like REQUEST operations. Rest of the transactions are synchronous anyway.

`module_slim_driver`(*__slim_driver*)
Helper macro for registering a SLIMbus driver

Parameters

`__slim_driver` slimbus_driver struct

Description

Helper macro for SLIMbus 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()`*

struct **`slim_framer`**
Represents SLIMbus framer. Every controller may have multiple framers. There is 1 active framer device responsible for clocking the bus. Manager is responsible for framer hand-over.

Definition

```
struct slim_framer {
  struct device          dev;
  struct slim_eaddr      e_addr;
  int rootfreq;
  int superfreq;
};
```

Members

`dev` Driver model representation of the device.

`e_addr` Enumeration address of the framer.

`rootfreq` Root Frequency at which the framer can run. This is maximum frequency ('clock gear 10') at which the bus can operate.

`superfreq` Superframes per root frequency. Every frame is 6144 bits.

struct **`slim_msg_txn`**
Message to be sent by the controller. This structure has packet header, payload and buffer to be filled (if any)

Definition

```
struct slim_msg_txn {
  u8 rl;
  u8 mt;
  u8 mc;
  u8 dt;
  u16 ec;
  u8 tid;
  u8 la;
  struct slim_val_inf     *msg;
  struct completion      *comp;
};
```

Members

rl Header field. remaining length.

mt Header field. Message type.

mc Header field. LSB is message code for type mt.

dt Header field. Destination type.

ec Element code. Used for elemental access APIs.

tid Transaction ID. Used for messages expecting response. (relevant for message-codes involving read operation)

la Logical address of the device this message is going to. (Not used when destination type is broadcast.)

msg Elemental access message to be read/written

comp completion if read/write is synchronous, used internally for tid based transactions.

enum **slim_clk_state**

Constants

SLIM_CLK_ACTIVE SLIMbus clock is active

SLIM_CLK_ENTERING_PAUSE SLIMbus clock pause sequence is being sent on the bus. If this succeeds, state changes to SLIM_CLK_PAUSED. If the transition fails, state changes back to SLIM_CLK_ACTIVE

SLIM_CLK_PAUSED SLIMbus controller clock has paused.

Description

maintaining current clock state.

struct **slim_sched**

Definition

```
struct slim_sched {
  enum slim_clk_state     clk_state;
  struct completion       pause_comp;
  struct mutex            m_reconf;
};
```

Members

clk_state Controller's clock state from enum slim_clk_state

pause_comp Signals completion of clock pause sequence. This is useful when client tries to call SLIMbus transaction when controller is entering clock pause.

m_reconf This mutex is held until current reconfiguration (data channel scheduling, message bandwidth reservation) is done. Message APIs can use the bus concurrently when this mutex is held since elemental access messages can be sent on the bus when reconfiguration is in progress.

struct **slim_controller**
Controls every instance of SLIMbus (similar to 'master' on SPI)

Definition

```
struct slim_controller {
  struct device          *dev;
  unsigned int           id;
  char name[SLIMBUS_NAME_SIZE];
  int min_cg;
  int max_cg;
  int clkgear;
  struct ida            laddr_ida;
  struct slim_framer    *a_framer;
  struct mutex          lock;
  struct list_head      devices;
  struct idr            tid_idr;
  spinlock_t txn_lock;
  struct slim_sched     sched;
  int (*xfer_msg)(struct slim_controller *ctrl, struct slim_msg_txn *tx);
  int (*set_laddr)(struct slim_controller *ctrl, struct slim_eaddr *ea, u8 laddr);
  int (*get_laddr)(struct slim_controller *ctrl, struct slim_eaddr *ea, u8 *laddr);
  int (*wakeup)(struct slim_controller *ctrl);
};
```

Members

dev Device interface to this driver

id Board-specific number identifier for this controller/bus

name Name for this controller

min_cg Minimum clock gear supported by this controller (default value: 1)

max_cg Maximum clock gear supported by this controller (default value: 10)

clkgear Current clock gear in which this bus is running

laddr_ida logical address id allocator

a_framer Active framer which is clocking the bus managed by this controller

lock Mutex protecting controller data structures

devices Slim device list

tid_idr tid id allocator

txn_lock Lock to protect table of transactions

sched scheduler structure used by the controller

xfer_msg Transfer a message on this controller (this can be a broadcast control/status message like data channel setup, or a unicast message like value element read/write.

set_laddr Setup logical address at laddr for the slave with elemental address e_addr. Drivers implementing controller will be expected to send unicast message to this device with its logical address.

get_laddr It is possible that controller needs to set fixed logical address table and get_laddr can be used in that case so that controller can do this assignment. Use case is when the master is on the remote processor side, who is resposible for allocating laddr.

wakeup This function pointer implements controller-specific procedure to wake it up from clock-pause. Framework will call this to bring the controller out of clock pause.

Description

'Manager device' is responsible for device management, bandwidth allocation, channel setup, and port associations per channel. Device management means Logical address assignment/removal based on enumeration (report-present, report-absent) of a device. Bandwidth allocation is done dynamically by the manager based on active channels on the bus, message-bandwidth requests made by SLIMbus devices. Based on current bandwidth usage, manager chooses a frequency to run the bus at (in steps of 'clock-gear', 1 through 10, each clock gear representing twice the frequency than the previous gear). Manager is also responsible for entering (and exiting) low-power-mode (known as 'clock pause'). Manager can do handover of framer if there are multiple framers on the bus and a certain usecase warrants using certain framer to avoid keeping previous framer being powered-on.

Controller here performs duties of the manager device, and 'interface device'. Interface device is responsible for monitoring the bus and reporting information such as loss-of-synchronization, data slot-collision.

int **slim_unregister_controller**(struct *slim_controller* * *ctrl*)
Controller tear-down.

Parameters

struct slim_controller * ctrl Controller to tear-down.

void **slim_report_absent**(struct *slim_device* * *sbdev*)
Controller calls this function when a device reports absent, OR when the device cannot be communicated with

Parameters

struct slim_device * sbdev Device that cannot be reached, or sent report absent

struct *slim_device* * **slim_get_device**(struct *slim_controller* * *ctrl*, struct *slim_eaddr* * *e_addr*)
get handle to a device.

Parameters

struct slim_controller * ctrl Controller on which this device will be added/queried

struct slim_eaddr * e_addr Enumeration address of the device to be queried

Return

pointer to a device if it has already reported. Creates a new device and returns pointer to it if the device has not yet enumerated.

int **slim_device_report_present**(struct *slim_controller* * *ctrl*, struct *slim_eaddr* * *e_addr*, u8 * *laddr*)
Report enumerated device.

Parameters

struct slim_controller * ctrl Controller with which device is enumerated.

struct slim_eaddr * e_addr Enumeration address of the device.

u8 * laddr Return logical address (if valid flag is false)

Description

Called by controller in response to REPORT_PRESENT. Framework will assign a logical address to this enumeration address. Function returns -EXFULL to indicate that all logical addresses are already taken.

int **slim_get_logical_addr**(struct *slim_device* * *sbdev*)
get/allocate logical address of a SLIMbus device.

Parameters

struct slim_device * sbdev client handle requesting the address.

Return

zero if a logical address is valid or a new logical address has been assigned. error code in case of error.

Clock-pause:

SLIMbus mandates that a reconfiguration sequence (known as clock-pause) be broadcast to all active devices on the bus before the bus can enter low-power mode. Controller uses this sequence when it decides to enter low-power mode so that corresponding clocks and/or power-rails can be turned off to save power. Clock-pause is exited by waking up framer device (if controller driver initiates exiting low power mode), or by toggling the data line (if a slave device wants to initiate it).

Clock-pause APIs:

int **slim_ctrl_clk_pause**(struct *slim_controller* * *ctrl*, bool *wakeup*, u8 *restart*)
Called by slimbus controller to enter/exit 'clock pause'

Parameters

struct slim_controller * ctrl controller requesting bus to be paused or woken up

bool wakeup Wakeup this controller from clock pause.

u8 restart Restart time value per spec used for clock pause. This value isn't used when controller is to be woken up.

Description

Slimbus specification needs this sequence to turn-off clocks for the bus. The sequence involves sending 3 broadcast messages (reconfiguration sequence) to inform all devices on the bus. To exit clock-pause, controller typically wakes up active framer device. This API executes clock pause reconfiguration sequence if wakeup is false. If wakeup is true, controller's wakeup is called. For entering clock-pause, -EBUSY is returned if a message txn in pending.

Messaging:

The framework supports regmap and read/write apis to exchange control-information with a SLIMbus device. APIs can be synchronous or asynchronous. The header file <linux/slimbus.h> has more documentation about messaging APIs.

Messaging APIs:

void **slim_msg_response**(struct *slim_controller* * *ctrl*, u8 * *reply*, u8 *tid*, u8 *len*)
Deliver Message response received from a device to the framework.

Parameters

struct slim_controller * ctrl Controller handle

u8 * reply Reply received from the device

u8 tid Transaction ID received with which framework can associate reply.

u8 len Length of the reply

Description

Called by controller to inform framework about the response received. This helps in making the API asynchronous, and controller-driver doesn't need to manage 1 more table other than the one managed by framework mapping TID with buffers

int **slim_do_transfer**(struct *slim_controller* * *ctrl*, struct *slim_msg_txn* * *txn*)
Process a SLIMbus-messaging transaction

Parameters

struct slim_controller * ctrl Controller handle

struct slim_msg_txn * txn Transaction to be sent over SLIMbus

Description

Called by controller to transmit messaging transactions not dealing with Interface/Value elements. (e.g. transmittting a message to assign logical address to a slave device

Return

-ETIMEDOUT: If transmission of this message timed out (e.g. due to bus lines not being clocked or driven by controller)

int **slim_xfer_msg**(struct *slim_device* * *sbdev*, struct *slim_val_inf* * *msg*, u8 *mc*)
Transfer a value info message on slim device

Parameters

struct slim_device * sbdev slim device to which this msg has to be transfered

struct slim_val_inf * msg value info message pointer

u8 mc message code of the message

Description

Called by drivers which want to transfer a vlaue or info elements.

Return

-ETIMEDOUT: If transmission of this message timed out

int **slim_read**(struct *slim_device* * *sdev*, u32 *addr*, size_t *count*, u8 * *val*)
Read SLIMbus value element

Parameters

struct slim_device * sdev client handle.

u32 addr address of value element to read.

size_t count number of bytes to read. Maximum bytes allowed are 16.

u8 * val will return what the value element value was

Return

-EINVAL for Invalid parameters, -ETIMEDOUT If transmission of this message timed out (e.g. due to bus lines not being clocked or driven by controller)

int **slim_readb**(struct *slim_device* * *sdev*, u32 *addr*)
Read byte from SLIMbus value element

Parameters

struct slim_device * sdev client handle.

u32 addr address in the value element to read.

Return

byte value of value element.

int **slim_write**(struct *slim_device* * *sdev*, u32 *addr*, size_t *count*, u8 * *val*)
Write SLIMbus value element

Parameters

struct slim_device * sdev client handle.

u32 addr address in the value element to write.

size_t count number of bytes to write. Maximum bytes allowed are 16.

u8 * val value to write to value element

Return

-EINVAL for Invalid parameters, -ETIMEDOUT If transmission of this message timed out (e.g. due to bus lines not being clocked or driven by controller)

int **slim_writeb**(struct *slim_device* * *sdev*, u32 *addr*, u8 *value*)
Write byte to SLIMbus value element

Parameters

struct slim_device * sdev client handle.

u32 addr address of value element to write.

u8 value value to write to value element

Return

-EINVAL for Invalid parameters, -ETIMEDOUT If transmission of this message timed out (e.g. due to bus lines not being clocked or driven by controller)

CHAPTER

THIRTYEIGHT

SOUNDWIRE DOCUMENTATION

SoundWire Subsystem Summary

SoundWire is a new interface ratified in 2015 by the MIPI Alliance. SoundWire is used for transporting data typically related to audio functions. SoundWire interface is optimized to integrate audio devices in mobile or mobile inspired systems.

SoundWire is a 2-pin multi-drop interface with data and clock line. It facilitates development of low cost, efficient, high performance systems. Broad level key features of SoundWire interface include:

1. Transporting all of payload data channels, control information, and setup commands over a single two-pin interface.
2. Lower clock frequency, and hence lower power consumption, by use of DDR (Dual Data Rate) data transmission.
3. Clock scaling and optional multiple data lanes to give wide flexibility in data rate to match system requirements.
4. Device status monitoring, including interrupt-style alerts to the Master.

The SoundWire protocol supports up to eleven Slave interfaces. All the interfaces share the common Bus containing data and clock line. Each of the Slaves can support up to 14 Data Ports. 13 Data Ports are dedicated to audio transport. Data Port0 is dedicated to transport of Bulk control information, each of the audio Data Ports (1..14) can support up to 8 Channels in transmit or receiving mode (typically fixed direction but configurable direction is enabled by the specification). Bandwidth restrictions to ~19.2..24.576Mbits/s don't however allow for 11*13*8 channels to be transmitted simultaneously.

Below figure shows an example of connectivity between a SoundWire Master and two Slave devices.

```
+---------------+                                       +---------------+
|               |                       Clock Signal    |               |
|    Master     |-------+-------------------------------|    Slave      |
|   Interface   |       |               Data Signal     |  Interface 1  |
|               |-------|-------+-----------------------|               |
+---------------+       |       |                       +---------------+
                        |       |
                        |       |
                        |       |
                     +--+-------+--+
                     |             |
                     |   Slave     |
                     | Interface 2 |
                     |             |
                     +-------------+
```

Terminology

The MIPI SoundWire specification uses the term 'device' to refer to a Master or Slave interface, which of course can be confusing. In this summary and code we use the term interface only to refer to the hardware. We follow the Linux device model by mapping each Slave interface connected on the bus as a device managed by a specific driver. The Linux SoundWire subsystem provides a framework to implement a SoundWire Slave driver with an API allowing 3rd-party vendors to enable implementation-defined functionality while common setup/configuration tasks are handled by the bus.

Bus: Implements SoundWire Linux Bus which handles the SoundWire protocol. Programs all the MIPI-defined Slave registers. Represents a SoundWire Master. Multiple instances of Bus may be present in a system.

Slave: Registers as SoundWire Slave device (Linux Device). Multiple Slave devices can register to a Bus instance.

Slave driver: Driver controlling the Slave device. MIPI-specified registers are controlled directly by the Bus (and transmitted through the Master driver/interface). Any implementation-defined Slave register is controlled by Slave driver. In practice, it is expected that the Slave driver relies on regmap and does not request direct register access.

Programming interfaces (SoundWire Master interface Driver)

SoundWire Bus supports programming interfaces for the SoundWire Master implementation and SoundWire Slave devices. All the code uses the "sdw" prefix commonly used by SoC designers and 3rd party vendors.

Each of the SoundWire Master interfaces needs to be registered to the Bus. Bus implements API to read standard Master MIPI properties and also provides callback in Master ops for Master driver to implement its own functions that provides capabilities information. DT support is not implemented at this time but should be trivial to add since capabilities are enabled with the `device_property_` API.

The Master interface along with the Master interface capabilities are registered based on board file, DT or ACPI.

Following is the Bus API to register the SoundWire Bus:

```
int sdw_add_bus_master(struct sdw_bus *bus)
{
        if (!bus->dev)
                return -ENODEV;

        mutex_init(&bus->lock);
        INIT_LIST_HEAD(&bus->slaves);

        /* Check ACPI for Slave devices */
        sdw_acpi_find_slaves(bus);

        /* Check DT for Slave devices */
        sdw_of_find_slaves(bus);

        return 0;
}
```

This will initialize sdw_bus object for Master device. "sdw_master_ops" and "sdw_master_port_ops" callback functions are provided to the Bus.

"sdw_master_ops" is used by Bus to control the Bus in the hardware specific way. It includes Bus control functions such as sending the SoundWire read/write messages on Bus, setting up clock frequency & Stream Synchronization Point (SSP). The "sdw_master_ops" structure abstracts the hardware details of the Master from the Bus.

"sdw_master_port_ops" is used by Bus to setup the Port parameters of the Master interface Port. Master interface Port register map is not defined by MIPI specification, so Bus calls the "sdw_master_port_ops" callback function to do Port operations like "Port Prepare", "Port Transport params set", "Port enable and disable". The implementation of the Master driver can then perform hardware-specific configurations.

Programming interfaces (SoundWire Slave Driver)

The MIPI specification requires each Slave interface to expose a unique 48-bit identifier, stored in 6 read-only dev_id registers. This dev_id identifier contains vendor and part information, as well as a field enabling to differentiate between identical components. An additional class field is currently unused. Slave driver is written for a specific vendor and part identifier, Bus enumerates the Slave device based on these two ids. Slave device and driver match is done based on these two ids . Probe of the Slave driver is called by Bus on successful match between device and driver id. A parent/child relationship is enforced between Master and Slave devices (the logical representation is aligned with the physical connectivity).

The information on Master/Slave dependencies is stored in platform data, board-file, ACPI or DT. The MIPI Software specification defines additional link_id parameters for controllers that have multiple Master interfaces. The dev_id registers are only unique in the scope of a link, and the link_id unique in the scope of a controller. Both dev_id and link_id are not necessarily unique at the system level but the parent/child information is used to avoid ambiguity.

```
static const struct sdw_device_id slave_id[] = {
        SDW_SLAVE_ENTRY(0x025d, 0x700, 0),
        {},
};
MODULE_DEVICE_TABLE(sdw, slave_id);

static struct sdw_driver slave_sdw_driver = {
        .driver = {
                   .name = "slave_xxx",
                   .pm = &slave_runtime_pm,
                   },
        .probe = slave_sdw_probe,
        .remove = slave_sdw_remove,
        .ops = &slave_slave_ops,
        .id_table = slave_id,
};
```

For capabilities, Bus implements API to read standard Slave MIPI properties and also provides callback in Slave ops for Slave driver to implement own function that provides capabilities information. Bus needs to know a set of Slave capabilities to program Slave registers and to control the Bus reconfigurations.

Future enhancements to be done

1. Bulk Register Access (BRA) transfers.
2. Multiple data lane support.

Links

SoundWire MIPI specification 1.1 is available at: https://members.mipi.org/wg/All-Members/document/70290

SoundWire MIPI DisCo (Discovery and Configuration) specification is available at: https://www.mipi.org/specifications/mipi-disco-soundwire

(publicly accessible with registration or directly accessible to MIPI members)

MIPI Alliance Manufacturer ID Page: mid.mipi.org