Linux Device Drivers

Table of Contents

Chapter 1. Driver Basics

Table of Contents

Driver Entry and Exit points

Name

module_init — driver initialization entry point

Synopsis

```
module_init (x);

x;
```

Arguments

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.

Name

module_exit — driver exit entry point

Synopsis

```
module_exit (x);

x;
```

Arguments

x

function to be run when driver is removed

Description

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

Atomic and pointer manipulation

Name

atomic_read — read atomic variable

Synopsis

```
int atomic_read (v);

const atomic_t *  v;
```

Arguments

v

pointer of type atomic_t

Description

Atomically reads the value of *v*.

Name

atomic_set — set atomic variable

Synopsis

```
void atomic_set (v,
                 i);

atomic_t *  v;
int         i;
```

Arguments

v

pointer of type atomic_t

i

required value

Description

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

Name

atomic_add — add integer to atomic variable

Synopsis

```
void atomic_add (i,
                 v);

int         i;
atomic_t *  v;
```

Arguments

i

integer value to add

v

pointer of type atomic_t

Description

Atomically adds *i* to *v*.

Name

atomic_sub — subtract integer from atomic variable

Synopsis

```
void atomic_sub (i,
                 v);

int         i;
atomic_t *  v;
```

Arguments

i

integer value to subtract

v

pointer of type atomic_t

Description

Atomically subtracts *i* from *v*.

Name

atomic_sub_and_test — subtract value from variable and test result

Synopsis

```
int atomic_sub_and_test (i,
                         v);

int         i;
atomic_t *  v;
```

Arguments

i

integer value to subtract

v

pointer of type atomic_t

Description

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

Name

atomic_inc — increment atomic variable

Synopsis

```
void atomic_inc (v);

atomic_t *  v;
```

Arguments

v

pointer of type atomic_t

Description

Atomically increments *v* by 1.

Name

atomic_dec — decrement atomic variable

Synopsis

```
void atomic_dec (v);

atomic_t *  v;
```

Arguments

v

pointer of type atomic_t

Description

Atomically decrements *v* by 1.

Name

atomic_dec_and_test — decrement and test

Synopsis

```
int atomic_dec_and_test (v);

atomic_t *  v;
```

Arguments

v

pointer of type atomic_t

Description

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

Name

atomic_inc_and_test — increment and test

Synopsis

```
int atomic_inc_and_test (v);

atomic_t *  v;
```

Arguments

v

pointer of type atomic_t

Description

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

Name

atomic_add_negative — add and test if negative

Synopsis

```
int atomic_add_negative (i,
```

```
                        v);

int         i;
atomic_t *  v;
```

Arguments

i

integer value to add

v

pointer of type atomic_t

Description

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

Name

atomic_add_return — add integer and return

Synopsis

```
int atomic_add_return (i,
                       v);

int         i;
atomic_t *  v;
```

Arguments

i

integer value to add

v

pointer of type atomic_t

Description

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

Name

atomic_sub_return — subtract integer and return

Synopsis

```
int atomic_sub_return (i,
                       v);

int         i;
atomic_t *  v;
```

Arguments

i

integer value to subtract

v

pointer of type atomic_t

Description

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

Name

atomic_add_unless — add unless the number is already a given value

Synopsis

```
int atomic_add_unless (v,
                       a,
                       u);

atomic_t *  v;
int         a;
int         u;
```

Arguments

v

pointer of type atomic_t

a

the amount to add to v...

u

...unless v is equal to u.

Description

Atomically adds *a* to *v*, so long as *v* was not already *u*. Returns non-zero if *v* was not *u*, and zero otherwise.

Name

atomic64_xchg — xchg atomic64 variable

Synopsis

```
u64 atomic64_xchg (ptr,
                   new_val);

atomic64_t *  ptr;
u64           new_val;
```

Arguments

ptr

pointer to type atomic64_t

new_val

value to assign

Description

Atomically xchgs the value of *ptr* to *new_val* and returns the old value.

Name

atomic64_set — set atomic64 variable

Synopsis

```
void atomic64_set (ptr,
                   new_val);

atomic64_t *  ptr;
u64           new_val;
```

Arguments

ptr

pointer to type atomic64_t

new_val

value to assign

Description

Atomically sets the value of *ptr* to *new_val*.

Name

atomic64_read — read atomic64 variable

Synopsis

```
u64 atomic64_read (ptr);

atomic64_t *  ptr;
```

Arguments

ptr

pointer to type atomic64_t

Description

Atomically reads the value of *ptr* and returns it.

Name

atomic64_add_return — add and return

Synopsis

```
u64 atomic64_add_return (delta,
                         ptr);

u64           delta;
atomic64_t *  ptr;
```

Arguments

delta

integer value to add

ptr

pointer to type atomic64_t

Description

Atomically adds *delta* to *ptr* and returns *delta* + **ptr*

Name

atomic64_add — add integer to atomic64 variable

Synopsis

```
void atomic64_add (delta,
                   ptr);

u64           delta;
atomic64_t *  ptr;
```

Arguments

delta

integer value to add

ptr

pointer to type atomic64_t

Description

Atomically adds *delta* to *ptr*.

Name

atomic64_sub — subtract the atomic64 variable

Synopsis

```
void atomic64_sub (delta,
                   ptr);

u64           delta;
atomic64_t *  ptr;
```

Arguments

delta

integer value to subtract

ptr

pointer to type atomic64_t

Description

Atomically subtracts *delta* from *ptr*.

Name

atomic64_sub_and_test — subtract value from variable and test result

Synopsis

```
int atomic64_sub_and_test (delta,
                           ptr);

u64           delta;
atomic64_t *  ptr;
```

Arguments

delta

integer value to subtract

ptr

pointer to type atomic64_t

Description

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

Name

atomic64_inc — increment atomic64 variable

Synopsis

```
void atomic64_inc (ptr);

atomic64_t *  ptr;
```

Arguments

ptr

pointer to type atomic64_t

Description

Atomically increments *ptr* by 1.

Name

atomic64_dec — decrement atomic64 variable

Synopsis

```
void atomic64_dec (ptr);

atomic64_t *  ptr;
```

Arguments

ptr

pointer to type atomic64_t

Description

Atomically decrements *ptr* by 1.

Name

atomic64_dec_and_test — decrement and test

Synopsis

```
int atomic64_dec_and_test (ptr);

atomic64_t *  ptr;
```

Arguments

ptr

pointer to type atomic64_t

Description

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

Name

atomic64_inc_and_test — increment and test

Synopsis

```
int atomic64_inc_and_test (ptr);

atomic64_t *  ptr;
```

Arguments

ptr

pointer to type atomic64_t

Description

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

Name

atomic64_add_negative — add and test if negative

Synopsis

```
int atomic64_add_negative (delta,
                           ptr);

u64           delta;
atomic64_t *  ptr;
```

Arguments

delta

integer value to add

ptr

pointer to type atomic64_t

Description

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

Name

/home/landley/linux/temp//arch/x86/include/asm/unaligned.h — Document generation inconsistency

Oops

Warning

The template for this document tried to insert the structured comment from the file `/home/landley/linux/temp//arch/x86/include/asm/unaligned.h` at this point, but none was found. This dummy section is inserted to allow generation to continue.

Delaying, scheduling, and timer routines

Name

struct task_cputime — collected CPU time counts

Synopsis

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

Members

utime

time spent in user mode, in cputime_t units

stime

time spent in kernel mode, in cputime_t units

sum_exec_runtime

total time spent on the CPU, in nanoseconds

Description

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

Name

struct thread_group_cputimer — thread group interval timer counts

Synopsis

```
struct thread_group_cputimer {
  struct task_cputime cputime;
  int running;
  spinlock_t lock;
};
```

Members

cputime

thread group interval timers.

running

non-zero when there are timers running and *cputime* receives updates.

lock

lock for fields in this struct.

Description

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

Name

pid_alive — check that a task structure is not stale

Synopsis

```
int pid_alive (p);

struct task_struct *  p;
```

Arguments

p

Task structure to be checked.

Description

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

Name

is_global_init — check if a task structure is init

Synopsis

```
int is_global_init (tsk);

struct task_struct *  tsk;
```

Arguments

tsk

Task structure to be checked.

Description

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

Name

wake_up_process — Wake up a specific process

Synopsis

```
int wake_up_process (p);

struct task_struct *  p;
```

Arguments

p

The process to be woken up.

Description

Attempt to wake up the nominated process and move it to the set of runnable processes. Returns 1 if the process was woken up, 0 if it was already running.

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

Name

preempt_notifier_register — tell me when current is being preempted & rescheduled

Synopsis

```
void preempt_notifier_register (notifier);

struct preempt_notifier *  notifier;
```

Arguments

notifier

notifier struct to register

Name

preempt_notifier_unregister — no longer interested in preemption notifications

Synopsis

```
void preempt_notifier_unregister (notifier);

struct preempt_notifier *  notifier;
```

Arguments

notifier

notifier struct to unregister

Description

This is safe to call from within a preemption notifier.

Name

__wake_up — wake up threads blocked on a waitqueue.

Synopsis

```
void __wake_up (q,
                mode,
                nr_exclusive,
                key);

wait_queue_head_t *  q;
unsigned int         mode;
int                  nr_exclusive;
void *               key;
```

Arguments

q

the waitqueue

mode

which threads

nr_exclusive

how many wake-one or wake-many threads to wake up

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.

Name

__wake_up_sync_key — wake up threads blocked on a waitqueue.

Synopsis

```
void __wake_up_sync_key (q,
                         mode,
                         nr_exclusive,
                         key);

wait_queue_head_t *  q;
unsigned int         mode;
int                  nr_exclusive;
void *               key;
```

Arguments

q

the waitqueue

mode

which threads

nr_exclusive

how many wake-one or wake-many threads to wake up

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.

Name

complete — signals a single thread waiting on this completion

Synopsis

```
void complete (x);

struct completion *  x;
```

Arguments

x

holds the state of this particular completion

Description

This will wake up a single thread waiting on this completion. Threads will be awakened in the same order in which they were queued.

See also `complete_all`, `wait_for_completion` and related routines.

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

Name

complete_all — signals all threads waiting on this completion

Synopsis

```
void complete_all (x);

struct completion *  x;
```

Arguments

x

holds the state of this particular completion

Description

This will wake up all threads waiting on this particular completion event.

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

Name

wait_for_completion — waits for completion of a task

Synopsis

```
void __sched wait_for_completion (x);

struct completion *  x;
```

Arguments

x

holds the state of this particular completion

Description

This waits to be signaled for completion of a specific task. It is NOT interruptible and there is no timeout.

See also similar routines (i.e. `wait_for_completion_timeout`) with timeout and interrupt capability. Also see `complete`.

Name

wait_for_completion_timeout — waits for completion of a task (w/timeout)

Synopsis

```
unsigned long __sched wait_for_completion_timeout (x,
                                                   timeout);

struct completion *  x;
unsigned long        timeout;
```

Arguments

x

holds the state of this particular completion

timeout

timeout value in jiffies

Description

This waits for either a completion of a specific task to be signaled or for a specified timeout to expire. The timeout is in jiffies. It is not interruptible.

Name

wait_for_completion_interruptible — waits for completion of a task (w/intr)

Synopsis

```
int __sched wait_for_completion_interruptible (x);

struct completion *  x;
```

Arguments

x

holds the state of this particular completion

Description

This waits for completion of a specific task to be signaled. It is interruptible.

Name

wait_for_completion_interruptible_timeout — waits for completion (w/(to,intr))

Synopsis

```
unsigned long __sched wait_for_completion_interruptible_timeout (x,
                                                                 timeout);

struct completion *  x;
unsigned long        timeout;
```

Arguments

x

holds the state of this particular completion

timeout

timeout value in jiffies

Description

This waits for either a completion of a specific task to be signaled or for a specified timeout to expire. It is interruptible. The timeout is in jiffies.

Name

wait_for_completion_killable — waits for completion of a task (killable)

Synopsis

```
int __sched wait_for_completion_killable (x);

struct completion *  x;
```

Arguments

x

holds the state of this particular completion

Description

This waits to be signaled for completion of a specific task. It can be interrupted by a kill signal.

Name

try_wait_for_completion — try to decrement a completion without blocking

Synopsis

```
bool try_wait_for_completion (x);

struct completion *  x;
```

Arguments

x

completion structure

Returns

0 if a decrement cannot be done without blocking 1 if a decrement succeeded.

If a completion is being used as a counting completion, attempt to decrement the counter without blocking. This enables us to avoid waiting if the resource the completion is protecting is not available.

Name

completion_done — Test to see if a completion has any waiters

Synopsis

```
bool completion_done (x);

struct completion *  x;
```

Arguments

x

completion structure

Returns

0 if there are waiters (`wait_for_completion` in progress) 1 if there are no waiters.

Name

task_nice — return the nice value of a given task.

Synopsis

```
int task_nice (p);

const struct task_struct *  p;
```

Arguments

p

the task in question.

Name

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

Synopsis

```
int sched_setscheduler (p,
                        policy,
                        param);

struct task_struct *  p;
int                   policy;
struct sched_param *  param;
```

Arguments

p

the task in question.

policy

new policy.

param

structure containing the new RT priority.

Description

NOTE that the task may be already dead.

Name

yield — yield the current processor to other threads.

Synopsis

```
void __sched yield (void);

void;
```

Arguments

void

no arguments

Description

This is a shortcut for kernel-space yielding - it marks the thread runnable and calls `sys_sched_yield`.

Name

__round_jiffies — function to round jiffies to a full second

Synopsis

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

unsigned long  j;
int            cpu;
```

Arguments

j

the time in (absolute) jiffies that should be rounded

cpu

the processor number on which the timeout will happen

Description

`__round_jiffies` rounds an absolute time in the future (in jiffies) up or down to (approximately) full

seconds. This is useful for timers for which the exact time they fire does not matter too much, as long as they fire approximately every X seconds.

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

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

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

Name

__round_jiffies_relative — function to round jiffies to a full second

Synopsis

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

unsigned long  j;
int            cpu;
```

Arguments

j

the time in (relative) jiffies that should be rounded

cpu

the processor number on which the timeout will happen

Description

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

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

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

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

Name

round_jiffies — function to round jiffies to a full second

Synopsis

```
unsigned long round_jiffies (j);

unsigned long  j;
```

Arguments

j

the time in (absolute) jiffies that should be rounded

Description

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

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

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

Name

round_jiffies_relative — function to round jiffies to a full second

Synopsis

```
unsigned long round_jiffies_relative (j);

unsigned long  j;
```

Arguments

j

the time in (relative) jiffies that should be rounded

Description

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

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

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

Name

__round_jiffies_up — function to round jiffies up to a full second

Synopsis

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

unsigned long  j;
int            cpu;
```

Arguments

j

the time in (absolute) jiffies that should be rounded

cpu

the processor number on which the timeout will happen

Description

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

Name

__round_jiffies_up_relative — function to round jiffies up to a full second

Synopsis

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

unsigned long  j;
```

```
int             cpu;
```

Arguments

j

the time in (relative) jiffies that should be rounded

cpu

the processor number on which the timeout will happen

Description

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

Name

round_jiffies_up — function to round jiffies up to a full second

Synopsis

```
unsigned long round_jiffies_up (j);

unsigned long  j;
```

Arguments

j

the time in (absolute) jiffies that should be rounded

Description

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

Name

round_jiffies_up_relative — function to round jiffies up to a full second

Synopsis

```
unsigned long round_jiffies_up_relative (j);

unsigned long  j;
```

Arguments

j

the time in (relative) jiffies that should be rounded

Description

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

Name

init_timer_key — initialize a timer

Synopsis

```
void init_timer_key (timer,
                     name,
                     key);

struct timer_list *      timer;
const char *             name;
struct lock_class_key *  key;
```

Arguments

timer

the timer to be initialized

name

name of the timer

key

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

Description

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

Name

mod_timer_pending — modify a pending timer's timeout

Synopsis

```
int mod_timer_pending (timer,
                       expires);

struct timer_list *  timer;
unsigned long        expires;
```

Arguments

timer

the pending timer to be modified

expires

new timeout in jiffies

Description

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

It is useful for unserialized use of timers.

Name

mod_timer — modify a timer's timeout

Synopsis

```
int mod_timer (timer,
               expires);

struct timer_list *  timer;
unsigned long        expires;
```

Arguments

timer

the timer to be modified

expires

new timeout in jiffies

Description

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

mod_timer(timer, expires) is equivalent to:

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

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

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

Name

mod_timer_pinned — modify a timer's timeout

Synopsis

```
int mod_timer_pinned (timer,
                      expires);

struct timer_list *  timer;
unsigned long        expires;
```

Arguments

timer

the timer to be modified

expires

new timeout in jiffies

Description

mod_timer_pinned is a way to update the expire field of an active timer (if the timer is inactive it will be activated) and not allow the timer to be migrated to a different CPU.

mod_timer_pinned(timer, expires) is equivalent to:

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

Name

add_timer — start a timer

Synopsis

```
void add_timer (timer);

struct timer_list *  timer;
```

Arguments

timer

the timer to be added

Description

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

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

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

Name

add_timer_on — start a timer on a particular CPU

Synopsis

```
void add_timer_on (timer,
                   cpu);

struct timer_list *  timer;
int                  cpu;
```

Arguments

timer

the timer to be added

cpu

the CPU to start it on

Description

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

Name

del_timer — deactive a timer.

Synopsis

```
int del_timer (timer);

struct timer_list *  timer;
```

Arguments

timer

the timer to be deactivated

Description

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

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

Name

try_to_del_timer_sync — Try to deactivate a timer

Synopsis

```
int try_to_del_timer_sync (timer);

struct timer_list *  timer;
```

Arguments

timer

timer do del

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.

It must not be called from interrupt contexts.

Name

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

Synopsis

```
int del_timer_sync (timer);

struct timer_list *  timer;
```

Arguments

timer

the timer to be deactivated

Description

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

Synchronization rules

Callers must prevent restarting of the timer, otherwise this function is meaningless. It must not be called from interrupt contexts. 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.

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

Name

schedule_timeout — sleep until timeout

Synopsis

```
signed long __sched schedule_timeout (timeout);

signed long  timeout;
```

Arguments

timeout

timeout value in jiffies

Description

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

You can set the task state as follows -

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

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

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

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

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

Name

msleep — sleep safely even with waitqueue interruptions

Synopsis

```
void msleep (msecs);

unsigned int  msecs;
```

Arguments

msecs

Time in milliseconds to sleep for

Name

msleep_interruptible — sleep waiting for signals

Synopsis

```
unsigned long msleep_interruptible (msecs);

unsigned int  msecs;
```

Arguments

msecs

Time in milliseconds to sleep for

High-resolution timers

Name

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

Synopsis

```
ktime_t ktime_set (secs,
                   nsecs);

const long           secs;
const unsigned long  nsecs;
```

Arguments

secs

seconds to set

nsecs

nanoseconds to set

Description

Return the ktime_t representation of the value

Name

ktime_sub — subtract two ktime_t variables

Synopsis

```
ktime_t ktime_sub (lhs,
                   rhs);

const ktime_t  lhs;
const ktime_t  rhs;
```

Arguments

lhs

minuend

rhs

subtrahend

Description

Returns the remainder of the substraction

Name

ktime_add — add two ktime_t variables

Synopsis

```
ktime_t ktime_add (add1,
                   add2);

const ktime_t  add1;
const ktime_t  add2;
```

Arguments

add1

addend1

add2

addend2

Description

Returns the sum of *add1* and *add2*.

Name

timespec_to_ktime — convert a timespec to ktime_t format

Synopsis

```
ktime_t timespec_to_ktime (ts);

const struct timespec  ts;
```

Arguments

ts

the timespec variable to convert

Description

Returns a ktime_t variable with the converted timespec value

Name

timeval_to_ktime — convert a timeval to ktime_t format

Synopsis

```
ktime_t timeval_to_ktime (tv);

const struct timeval  tv;
```

Arguments

tv

the timeval variable to convert

Description

Returns a ktime_t variable with the converted timeval value

Name

ktime_to_timespec — convert a ktime_t variable to timespec format

Synopsis

```
struct timespec ktime_to_timespec (kt);

const ktime_t kt;
```

Arguments

kt

the ktime_t variable to convert

Description

Returns the timespec representation of the ktime value

Name

ktime_to_timeval — convert a ktime_t variable to timeval format

Synopsis

```
struct timeval ktime_to_timeval (kt);

const ktime_t kt;
```

Arguments

kt

the ktime_t variable to convert

Description

Returns the timeval representation of the ktime value

Name

ktime_to_ns — convert a ktime_t variable to scalar nanoseconds

Synopsis

```
s64 ktime_to_ns (kt);

const ktime_t  kt;
```

Arguments

kt

the ktime_t variable to convert

Description

Returns the scalar nanoseconds representation of *kt*

Name

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

Synopsis

```
int ktime_equal (cmp1,
                 cmp2);

const ktime_t  cmp1;
const ktime_t  cmp2;
```

Arguments

cmp1

comparable1

cmp2

comparable2

Description

Compare two ktime_t variables, returns 1 if equal

Name

struct hrtimer — the basic hrtimer structure

Synopsis

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

Members

node

red black tree node for time ordered insertion

_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)

start_pid

timer statistics field to store the pid of the task which started the timer

start_site

timer statistics field to store the site where the timer was started

start_comm[16]

timer statistics field to store the name of the process which started the timer

Description

The hrtimer structure must be initialized by `hrtimer_init`

Name

struct hrtimer_sleeper — simple sleeper structure

Synopsis

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

Members

timer

embedded timer structure

task

task to wake up

Description

task is set to NULL, when the timer expires.

Name

struct hrtimer_clock_base — the timer base for a specific clock

Synopsis

```
struct hrtimer_clock_base {
```

```
  struct hrtimer_cpu_base * cpu_base;
  clockid_t index;
  struct rb_root active;
  struct rb_node * first;
  ktime_t resolution;
  ktime_t (* get_time) (void);
  ktime_t softirq_time;
#ifdef CONFIG_HIGH_RES_TIMERS
  ktime_t offset;
#endif
};
```

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.

active

red black tree root node for the active timers

first

pointer to the timer node which expires first

resolution

the resolution of the clock, in nanoseconds

get_time

function to retrieve the current time of the clock

softirq_time

the time when running the hrtimer queue in the softirq

offset

offset of this clock to the monotonic base

Name

ktime_add_ns — Add a scalar nanoseconds value to a ktime_t variable

Synopsis

```
ktime_t ktime_add_ns (kt,
                      nsec);

const ktime_t  kt;
u64            nsec;
```

Arguments

kt

addend

nsec

the scalar nsec value to add

Description

Returns the sum of kt and nsec in ktime_t format

Name

ktime_sub_ns — Subtract a scalar nanoseconds value from a ktime_t variable

Synopsis

```
ktime_t ktime_sub_ns (kt,
                      nsec);

const ktime_t  kt;
u64            nsec;
```

Arguments

kt

minuend

nsec

the scalar nsec value to subtract

Description

Returns the subtraction of *nsec* from *kt* in ktime_t format

---

Name

hrtimer_forward — forward the timer expiry

Synopsis

```
u64 hrtimer_forward (timer,
                     now,
                     interval);

struct hrtimer *  timer;
ktime_t           now;
ktime_t           interval;
```

Arguments

timer

hrtimer to forward

now

forward past this time

interval

the interval to forward

Description

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

Name

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

Synopsis

```
int hrtimer_start_range_ns (timer,
                            tim,
                            delta_ns,
                            mode);

struct hrtimer *         timer;
ktime_t                  tim;
unsigned long            delta_ns;
const enum hrtimer_mode  mode;
```

Arguments

timer

the timer to be added

tim

expiry time

delta_ns

"slack" range for the timer

mode

expiry mode: absolute (HRTIMER_ABS) or relative (HRTIMER_REL)

Returns

0 on success 1 when the timer was active

Name

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

Synopsis

```
int hrtimer_start (timer,
                   tim,
                   mode);

struct hrtimer *         timer;
ktime_t                  tim;
const enum hrtimer_mode  mode;
```

Arguments

timer

the timer to be added

tim

expiry time

mode

expiry mode: absolute (HRTIMER_ABS) or relative (HRTIMER_REL)

Returns

0 on success 1 when the timer was active

Name

hrtimer_try_to_cancel — try to deactivate a timer

Synopsis

```
int hrtimer_try_to_cancel (timer);

struct hrtimer *  timer;
```

Arguments

timer

hrtimer to stop

Returns

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

---

Name

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

Synopsis

```
int hrtimer_cancel (timer);

struct hrtimer *  timer;
```

Arguments

timer

the timer to be cancelled

Returns

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

Name

hrtimer_get_remaining — get remaining time for the timer

Synopsis

```
ktime_t hrtimer_get_remaining (timer);

const struct hrtimer *  timer;
```

Arguments

timer

the timer to read

Name

hrtimer_init — initialize a timer to the given clock

Synopsis

```
void hrtimer_init (timer,
                   clock_id,
                   mode);

struct hrtimer *   timer;
clockid_t          clock_id;
enum hrtimer_mode  mode;
```

Arguments

timer

the timer to be initialized

clock_id

the clock to be used

mode

timer mode abs/rel

Name

hrtimer_get_res — get the timer resolution for a clock

Synopsis

```
int hrtimer_get_res (which_clock,
                     tp);

const clockid_t    which_clock;
struct timespec *  tp;
```

Arguments

which_clock

which clock to query

tp

pointer to timespec variable to store the resolution

Description

Store the resolution of the clock selected by *which_clock* in the variable pointed to by *tp*.

Name

schedule_hrtimeout_range — sleep until timeout

Synopsis

```
int __sched schedule_hrtimeout_range (expires,
                                      delta,
                                      mode);

ktime_t *                expires;
unsigned long            delta;
const enum hrtimer_mode  mode;
```

Arguments

expires

timeout value (ktime_t)

delta

slack in expires timeout (ktime_t)

mode

timer mode, HRTIMER_MODE_ABS or HRTIMER_MODE_REL

Description

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

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

You can set the task state as follows -

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

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

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

Returns 0 when the timer has expired otherwise -EINTR

Name

schedule_hrtimeout — sleep until timeout

Synopsis

```
int __sched schedule_hrtimeout (expires,
                                mode);

ktime_t *                expires;
const enum hrtimer_mode  mode;
```

Arguments

expires

timeout value (ktime_t)

mode

timer mode, HRTIMER_MODE_ABS or HRTIMER_MODE_REL

Description

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

You can set the task state as follows -

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

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

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

Returns 0 when the timer has expired otherwise -EINTR

Workqueues and Kevents

Name

queue_work — queue work on a workqueue

Synopsis

```
int queue_work (wq,
                work);

struct workqueue_struct *  wq;
struct work_struct *       work;
```

Arguments

`wq`

workqueue to use

`work`

work to queue

Description

Returns 0 if *`work`* was already on a queue, non-zero otherwise.

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

Name

queue_work_on — queue work on specific cpu

Synopsis

```
int queue_work_on (cpu,
                   wq,
                   work);

int                         cpu;
struct workqueue_struct *   wq;
struct work_struct *        work;
```

Arguments

cpu

CPU number to execute work on

wq

workqueue to use

work

work to queue

Description

Returns 0 if *work* was already on a queue, non-zero otherwise.

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

Name

queue_delayed_work — queue work on a workqueue after delay

Synopsis

```
int queue_delayed_work (wq,
                        dwork,
                        delay);

struct workqueue_struct *  wq;
struct delayed_work *      dwork;
unsigned long              delay;
```

Arguments

wq

workqueue to use

dwork

delayable work to queue

delay

number of jiffies to wait before queueing

Description

Returns 0 if *work* was already on a queue, non-zero otherwise.

Name

queue_delayed_work_on — queue work on specific CPU after delay

Synopsis

```
int queue_delayed_work_on (cpu,
                           wq,
                           dwork,
                           delay);

int                        cpu;
struct workqueue_struct *  wq;
struct delayed_work *      dwork;
unsigned long              delay;
```

Arguments

cpu

CPU number to execute work on

wq

workqueue to use

dwork

work to queue

delay

number of jiffies to wait before queueing

Description

Returns 0 if *work* was already on a queue, non-zero otherwise.

Name

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

Synopsis

```
void flush_workqueue (wq);

struct workqueue_struct *  wq;
```

Arguments

wq

workqueue to flush

Description

Forces execution of the workqueue and blocks until its completion. This is typically used in driver shutdown handlers.

We sleep until all works which were queued on entry have been handled, but we are not livelocked by new incoming ones.

This function used to run the workqueues itself. Now we just wait for the helper threads to do it.

Name

flush_work — block until a work_struct's callback has terminated

Synopsis

```
int flush_work (work);

struct work_struct *  work;
```

Arguments

work

the work which is to be flushed

Description

Returns false if *work* has already terminated.

It is expected that, prior to calling `flush_work`, the caller has arranged for the work to not be requeued, otherwise it doesn't make sense to use this function.

Name

cancel_work_sync — block until a work_struct's callback has terminated

Synopsis

```
int cancel_work_sync (work);

struct work_struct *  work;
```

Arguments

work

the work which is to be flushed

Description

Returns true if *work* was pending.

`cancel_work_sync` will cancel the work if it is queued. If the work's callback appears to be running, `cancel_work_sync` will block until it has completed.

It is possible to use this function if the work re-queues itself. It can cancel the work even if it migrates to another workqueue, however in that case it only guarantees that work->`func` has completed on the last queued workqueue.

cancel_work_sync(delayed_work->work) should be used only if ->timer is not pending, otherwise it goes into a busy-wait loop until the timer expires.

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

Name

cancel_delayed_work_sync — reliably kill off a delayed work.

Synopsis

```
int cancel_delayed_work_sync (dwork);

struct delayed_work *  dwork;
```

Arguments

dwork

the delayed work struct

Description

Returns true if *dwork* was pending.

It is possible to use this function if *dwork* rearms itself via `queue_work` or `queue_delayed_work`. See also the comment for `cancel_work_sync`.

Name

schedule_work — put work task in global workqueue

Synopsis

```
int schedule_work (work);

struct work_struct *  work;
```

Arguments

work

job to be done

Description

Returns zero if *work* was already on the kernel-global workqueue and non-zero otherwise.

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

Name

schedule_delayed_work — put work task in global workqueue after delay

Synopsis

```
int schedule_delayed_work (dwork,
                           delay);

struct delayed_work *  dwork;
unsigned long          delay;
```

Arguments

dwork

job to be done

delay

number of jiffies to wait or 0 for immediate execution

Description

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

Name

flush_delayed_work — block until a dwork_struct's callback has terminated

Synopsis

```
void flush_delayed_work (dwork);

struct delayed_work *  dwork;
```

Arguments

dwork

the delayed work which is to be flushed

Description

Any timeout is cancelled, and any pending work is run immediately.

Name

schedule_delayed_work_on — queue work in global workqueue on CPU after delay

Synopsis

```
int schedule_delayed_work_on (cpu,
                              dwork,
                              delay);

int                   cpu;
struct delayed_work *  dwork;
unsigned long          delay;
```

Arguments

cpu

cpu to use

dwork

job to be done

delay

number of jiffies to wait

Description

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

Name

execute_in_process_context — reliably execute the routine with user context

Synopsis

```
int execute_in_process_context (fn,
                                ew);

work_func_t            fn;
struct execute_work *  ew;
```

Arguments

fn

the function to execute

ew

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

Description

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

Returns

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

---

Name

destroy_workqueue — safely terminate a workqueue

Synopsis

```
void destroy_workqueue (wq);

struct workqueue_struct *  wq;
```

Arguments

wq

target workqueue

Description

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

Name

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

Synopsis

```
long work_on_cpu (cpu,
                  fn,
                  arg);

unsigned int  cpu;
long (*       fn(void *);
void *        arg;
```

Arguments

cpu

the cpu to run on

fn

the function to run

arg

the function arg

Description

This will return the value *fn* returns. 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.

Internal Functions

Name

reparent_to_kthreadd — Reparent the calling kernel thread to kthreadd

Synopsis

```
void reparent_to_kthreadd (void);

 void;
```

Arguments

void

no arguments

Description

If a kernel thread is launched as a result of a system call, or if it ever exits, it should generally reparent itself to kthreadd so it isn't in the way of other processes and is correctly cleaned up on exit.

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

NOTE that `reparent_to_kthreadd` gives the caller full capabilities.

Name

sys_tgkill — send signal to one specific thread

Synopsis

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

pid_t  tgid;
pid_t  pid;
int    sig;
```

Arguments

tgid

the thread group ID of the thread

pid

the PID of the thread

sig

signal to be sent

Description

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

Name

kthread_run — create and wake a thread.

Synopsis

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

 threadfn;
 data;
 namefmt;
 ...;
```

Arguments

threadfn

the function to run until signal_pending(current).

data

data ptr for *threadfn*.

namefmt

printf-style name for the thread.

...

variable arguments

Description

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

Name

kthread_should_stop — should this kthread return now?

Synopsis

```
int kthread_should_stop (void);

 void;
```

Arguments

void

no arguments

Description

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

Name

kthread_create — create a kthread.

Synopsis

```
struct task_struct * kthread_create (threadfn,
                                     data,
                                     namefmt[],
                                     ...);

int (*      threadfn(void *data);
void *      data;
const char  namefmt[];
            ...;
```

Arguments

`threadfn`

the function to run until signal_pending(current).

`data`

data ptr for *`threadfn`*.

`namefmt[]`

printf-style name for the thread.

`...`

variable arguments

Description

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`, `kthread_create_on_cpu`.

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 noone 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).

Name

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

Synopsis

```
void kthread_bind (k,
                   cpu);

struct task_struct *  k;
unsigned int          cpu;
```

Arguments

k

thread created by `kthread_create`.

cpu

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

Description

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

Name

kthread_stop — stop a thread created by `kthread_create`.

Synopsis

```
int kthread_stop (k);

struct task_struct *  k;
```

Arguments

k

thread created by `kthread_create`.

Description

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

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

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

Kernel objects manipulation

Name

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

Synopsis

```
char * kobject_get_path (kobj,
                         gfp_mask);

struct kobject *  kobj;
gfp_t             gfp_mask;
```

Arguments

`kobj`

kobject in question, with which to build the path

`gfp_mask`

the allocation type used to allocate the path

Description

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

Name

kobject_set_name — Set the name of a kobject

Synopsis

```
int kobject_set_name (kobj,
                      fmt,
                      ...);

struct kobject *  kobj;
const char *      fmt;
                  ...;
```

Arguments

kobj

struct kobject to set the name of

fmt

format string used to build the name

...

variable arguments

Description

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

Name

kobject_init — initialize a kobject structure

Synopsis

```
void kobject_init (kobj,
                   ktype);

struct kobject *    kobj;
struct kobj_type *  ktype;
```

Arguments

kobj

pointer to the kobject to initialize

ktype

pointer to the ktype for this kobject.

Description

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

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

Name

kobject_add — the main kobject add function

Synopsis

```
int kobject_add (kobj,
                 parent,
                 fmt,
                 ...);

struct kobject *  kobj;
struct kobject *  parent;
const char *      fmt;
                  ...;
```

Arguments

kobj

the kobject to add

parent

pointer to the parent of the kobject.

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 associted with the kset assigned to this kobject. If no kset is assigned to

the kobject, then the kobject will be located in the root of the sysfs tree.

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

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

Name

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

Synopsis

```
int kobject_init_and_add (kobj,
                          ktype,
                          parent,
                          fmt,
                          ...);

struct kobject *    kobj;
struct kobj_type *  ktype;
struct kobject *    parent;
const char *        fmt;
                    ...;
```

Arguments

kobj

pointer to the kobject to initialize

ktype

pointer to the ktype for this kobject.

parent

pointer to the parent of this kobject.

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.

Name

kobject_rename — change the name of an object

Synopsis

```
int kobject_rename (kobj,
                    new_name);

struct kobject *  kobj;
const char *      new_name;
```

Arguments

kobj

object in question.

new_name

object's new name

Description

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

Name

kobject_del — unlink kobject from hierarchy.

Synopsis

```
void kobject_del (kobj);

struct kobject *  kobj;
```

Arguments

kobj

object.

Name

kobject_get — increment refcount for object.

Synopsis

```
struct kobject * kobject_get (kobj);

struct kobject *  kobj;
```

Arguments

kobj

object.

Name

kobject_put — decrement refcount for object.

Synopsis

```
void kobject_put (kobj);

struct kobject *  kobj;
```

Arguments

kobj

object.

Description

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

Name

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

Synopsis

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

const char *      name;
struct kobject *  parent;
```

Arguments

name

the name for the kset

parent

the parent kobject of this kobject, if any.

Description

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

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

Name

kset_register — initialize and add a kset.

Synopsis

```
int kset_register (k);

struct kset *  k;
```

Arguments

k

kset.

Name

kset_unregister — remove a kset.

Synopsis

```
void kset_unregister (k);

struct kset *  k;
```

Arguments

k

kset.

Name

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

Synopsis

```
struct kset * kset_create_and_add (name,
                                   uevent_ops,
                                   parent_kobj);

const char *              name;
struct kset_uevent_ops *  uevent_ops;
struct kobject *          parent_kobj;
```

Arguments

name

the name for the kset

uevent_ops

a struct kset_uevent_ops for the kset

parent_kobj

the parent kobject of this kset, if any.

Description

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

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

Kernel utility functions

Name

upper_32_bits — return bits 32-63 of a number

Synopsis

```
upper_32_bits (n);

 n;
```

Arguments

n

the number we're accessing

Description

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

Name

lower_32_bits — return bits 0-31 of a number

Synopsis

```
lower_32_bits (n);

 n;
```

Arguments

n

the number we're accessing

Name

might_sleep — annotation for functions that can sleep

Synopsis

```
might_sleep ();
```

Arguments

None

Description

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

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

Name

trace_printk — printf formatting in the ftrace buffer

Synopsis

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

fmt;
args...;
```

Arguments

fmt

the printf format for printing

args...

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.

Name

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

Synopsis

```
clamp (val,
       min,
       max);

 val;
 min;
 max;
```

Arguments

val

current value

min

minimum allowable value

max

maximum allowable value

Description

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

Name

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

Synopsis

```
clamp_t (type,
         val,
```

min,
max);

type;
val;
min;
max;

Arguments

type

the type of variable to use

val

current value

min

minimum allowable value

max

maximum allowable value

Description

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

Name

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

Synopsis

clamp_val (*val*,
min,
max);

val;
min;
max;

Arguments

val

current value

min

minimum allowable value

max

maximum allowable value

Description

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

Name

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

Synopsis

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

 ptr;
 type;
 member;
```

Arguments

ptr

the pointer to the member.

type

the type of the container struct this is embedded in.

member

the name of the member within the struct.

Name

printk — print a kernel message

Synopsis

```
int printk (fmt,
            ...);

const char *  fmt;
              ...;
```

Arguments

fmt

format string

...

variable arguments

Description

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

We try to grab the console_sem. 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 `release_console_sem` and will send it to the consoles before releasing the semaphore.

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

See also

printf(3)

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

Name

acquire_console_sem — lock the console system for exclusive use.

Synopsis

```
void acquire_console_sem (void);

 void;
```

Arguments

void

no arguments

Description

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

Can sleep, returns nothing.

Name

release_console_sem — unlock the console system

Synopsis

```
void release_console_sem (void);
```

void;

Arguments

void

no arguments

Description

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

While the semaphore was held, console output may have been buffered by `printk`. If this is the case, `release_console_sem` emits the output prior to releasing the semaphore.

If there is output waiting for klogd, we wake it up.

`release_console_sem` may be called from any context.

Name

console_conditional_schedule — yield the CPU if required

Synopsis

```
void __sched console_conditional_schedule (void);

 void;
```

Arguments

void

no arguments

Description

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

Must be called within `acquire_console_sem`.

Name

printk_timed_ratelimit — caller-controlled printk ratelimiting

Synopsis

```
bool printk_timed_ratelimit (caller_jiffies,
                             interval_msecs);

unsigned long *  caller_jiffies;
unsigned int     interval_msecs;
```

Arguments

caller_jiffies

pointer to caller's state

interval_msecs

minimum interval between prints

Description

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

Name

panic — halt the system

Synopsis

```
NORET_TYPE void panic (fmt,
                       ...);

const char *  fmt;
              ...;
```

Arguments

fmt

The text string to print

...

variable arguments

Description

Display a message, then perform cleanups.

This function never returns.

Name

emergency_restart — reboot the system

Synopsis

```
void emergency_restart (void);

 void;
```

Arguments

void

no arguments

Description

Without shutting down any hardware or taking any locks reboot the system. This is called when we know we are in trouble so this is our best effort to reboot. This is safe to call in interrupt context.

Name

kernel_restart — reboot the system

Synopsis

```
void kernel_restart (cmd);

char *  cmd;
```

Arguments

cmd

pointer to buffer containing command to execute for restart or NULL

Description

Shutdown everything and perform a clean reboot. This is not safe to call in interrupt context.

Name

kernel_halt — halt the system

Synopsis

```
void kernel_halt (void);

 void;
```

Arguments

void

no arguments

Description

Shutdown everything and perform a clean system halt.

Name

kernel_power_off — power_off the system

Synopsis

```
void kernel_power_off (void);

 void;
```

Arguments

void

no arguments

Description

Shutdown everything and perform a clean system power_off.

Name

orderly_poweroff — Trigger an orderly system poweroff

Synopsis

```
int orderly_poweroff (force);

bool  force;
```

Arguments

force

force poweroff if command execution fails

Description

This may be called from any context to trigger a system shutdown. If the orderly shutdown fails, it will force an immediate shutdown.

Name

synchronize_rcu — wait until a grace period has elapsed.

Synopsis

```
void synchronize_rcu (void);

 void;
```

Arguments

void

no arguments

Description

Control will return to the caller some time after a full grace period has elapsed, in other words after all currently executing RCU read-side critical sections have completed. RCU read-side critical sections are delimited by `rcu_read_lock` and `rcu_read_unlock`, and may be nested.

Name

synchronize_sched — wait until an rcu-sched grace period has elapsed.

Synopsis

```
void synchronize_sched (void);

 void;
```

Arguments

void

no arguments

Description

Control will return to the caller some time after a full rcu-sched grace period has elapsed, in other words after all currently executing rcu-sched read-side critical sections have completed. These read-side critical sections are delimited by `rcu_read_lock_sched` and `rcu_read_unlock_sched`, and may be nested. Note that `preempt_disable`, `local_irq_disable`, and so on may be used in place of `rcu_read_lock_sched`.

This means that all preempt_disable code sequences, including NMI and 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.

This primitive provides the guarantees made by the (now removed) `synchronize_kernel` API. In contrast, `synchronize_rcu` only guarantees that `rcu_read_lock` sections will have completed. In "classic RCU", these two guarantees happen to be one and the same, but can differ in realtime RCU implementations.

Name

synchronize_rcu_bh — wait until an rcu_bh grace period has elapsed.

Synopsis

```
void synchronize_rcu_bh (void);

 void;
```

Arguments

void

no arguments

Description

Control will return to the caller some time after a full rcu_bh grace period has elapsed, in other words after all currently executing rcu_bh read-side critical sections have completed. RCU read-side critical sections are delimited by `rcu_read_lock_bh` and `rcu_read_unlock_bh`, and may be nested.

Device Resource Management

Name

devres_alloc — Allocate device resource data

Synopsis

```
void * devres_alloc (release,
                     size,
                     gfp);

dr_release_t  release;
size_t        size;
gfp_t         gfp;
```

Arguments

release

Release function devres will be associated with

size

Allocation size

gfp

Allocation flags

Description

Allocate devres of *size* bytes. The allocated area is zeroed, then associated with *release*. The returned pointer can be passed to other devres_*() functions.

RETURNS

Pointer to allocated devres on success, NULL on failure.

Name

devres_free — Free device resource data

Synopsis

```
void devres_free (res);

void * res;
```

Arguments

res

Pointer to devres data to free

Description

Free devres created with `devres_alloc`.

Name

devres_add — Register device resource

Synopsis

```
void devres_add (dev,
                 res);

struct device *  dev;
void *           res;
```

Arguments

dev

Device to add resource to

res

Resource to register

Description

Register devres *res* to *dev*. *res* should have been allocated using devres_alloc. On driver detach, the associated release function will be invoked and devres will be freed automatically.

Name

devres_find — Find device resource

Synopsis

```
void * devres_find (dev,
                    release,
                    match,
                    match_data);

struct device *  dev;
dr_release_t     release;
dr_match_t       match;
void *           match_data;
```

Arguments

dev

Device to lookup resource from

release

Look for resources associated with this release function

match

Match function (optional)

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.

RETURNS

Pointer to found devres, NULL if not found.

Name

devres_get — Find devres, if non-existent, add one atomically

Synopsis

```
void * devres_get (dev,
                   new_res,
                   match,
                   match_data);

struct device *  dev;
void *           new_res;
dr_match_t       match;
void *           match_data;
```

Arguments

dev

Device to lookup or add devres for

new_res

Pointer to new initialized devres to add if not found

match

Match function (optional)

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.

RETURNS

Pointer to found or added devres.

Name

devres_remove — Find a device resource and remove it

Synopsis

```
void * devres_remove (dev,
                      release,
                      match,
                      match_data);

struct device *  dev;
dr_release_t     release;
dr_match_t       match;
void *           match_data;
```

Arguments

dev

Device to find resource from

release

Look for resources associated with this release function

match

Match function (optional)

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.

RETURNS

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

Name

devres_destroy — Find a device resource and destroy it

Synopsis

```
int devres_destroy (dev,
                    release,
                    match,
                    match_data);

struct device *  dev;
dr_release_t     release;
dr_match_t       match;
void *           match_data;
```

Arguments

dev

Device to find resource from

release

Look for resources associated with this release function

match

Match function (optional)

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.

RETURNS

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

Name

devres_open_group — Open a new devres group

Synopsis

```
void * devres_open_group (dev,
                          id,
                          gfp);

struct device *  dev;
void *           id;
gfp_t            gfp;
```

Arguments

dev

Device to open devres group for

id

Separator ID

gfp

Allocation flags

Description

Open a new devres group for *dev* with *id*. For *id*, using a pointer to an object which won't be used for another group is recommended. If *id* is NULL, address-wise unique ID is created.

RETURNS

ID of the new group, NULL on failure.

Name

devres_close_group — Close a devres group

Synopsis

```
void devres_close_group (dev,
                         id);

struct device *  dev;
void *           id;
```

Arguments

dev

Device to close devres group for

id

ID of target group, can be NULL

Description

Close the group identified by *id*. If *id* is NULL, the latest open group is selected.

Name

devres_remove_group — Remove a devres group

Synopsis

```
void devres_remove_group (dev,
                          id);

struct device *  dev;
void *           id;
```

Arguments

dev

Device to remove group for

id

ID of target group, can be NULL

Description

Remove the group identified by *id*. If *id* is NULL, the latest open group is selected. Note that removing a group doesn't affect any other resources.

Name

devres_release_group — Release resources in a devres group

Synopsis

```
int devres_release_group (dev,
                          id);

struct device *  dev;
void *           id;
```

Arguments

dev

Device to release group for

id

ID of target group, can be NULL

Description

Release all resources in the group identified by *id*. If *id* is NULL, the latest open group is selected. The selected group and groups properly nested inside the selected group are removed.

RETURNS

The number of released non-group resources.

Name

devm_kzalloc — Resource-managed kzalloc

Synopsis

```
void * devm_kzalloc (dev,
                     size,
                     gfp);

struct device *  dev;
```

```
size_t          size;
gfp_t           gfp;
```

Arguments

dev

Device to allocate memory for

size

Allocation size

gfp

Allocation gfp flags

Description

Managed kzalloc. Memory allocated with this function is automatically freed on driver detach. Like all other devres resources, guaranteed alignment is unsigned long long.

RETURNS

Pointer to allocated memory on success, NULL on failure.

Name

devm_kfree — Resource-managed kfree

Synopsis

```
void devm_kfree (dev,
                 p);

struct device *  dev;
void *           p;
```

Arguments

dev

Device this memory belongs to

p

Memory to free

Description

Free memory allocated with `dev_kzalloc`.

Chapter 2. Device drivers infrastructure

Table of Contents

Device Drivers Base

Name

driver_for_each_device — Iterator for devices bound to a driver.

Synopsis

```
int driver_for_each_device (drv,
                            start,
                            data,
                            fn);

struct device_driver *  drv;
struct device *         start;
void *                  data;
int (*                  fn(struct device *, void *);
```

Arguments

drv

Driver we're iterating.

start

Device to begin with

data

Data to pass to the callback.

fn

Function to call for each device.

Description

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

Name

driver_find_device — device iterator for locating a particular device.

Synopsis

```
struct device * driver_find_device (drv,
                                    start,
                                    data,
                                    match);

struct device_driver *  drv;
struct device *         start;
void *                  data;
int (*                  match(struct device *dev, void *data);
```

Arguments

drv

The device's driver

start

Device to begin with

data

Data to pass to match function

match

Callback function to check device

Description

This is similar to the `driver_for_each_device` function above, but it returns a reference to a device that is 'found' for later use, as determined by the *match* callback.

The callback should return 0 if the device doesn't match and non-zero if it does. If the callback returns non-zero, this function will return to the caller and not iterate over any more devices.

Name

driver_create_file — create sysfs file for driver.

Synopsis

```
int driver_create_file (drv,
                        attr);

struct device_driver *     drv;
struct driver_attribute *  attr;
```

Arguments

drv

driver.

attr

driver attribute descriptor.

Name

driver_remove_file — remove sysfs file for driver.

Synopsis

```
void driver_remove_file (drv,
                         attr);

struct device_driver *     drv;
struct driver_attribute *  attr;
```

Arguments

drv

driver.

attr

driver attribute descriptor.

Name

driver_add_kobj — add a kobject below the specified driver

Synopsis

```
int driver_add_kobj (drv,
                     kobj,
                     fmt,
                     ...);

struct device_driver *  drv;
struct kobject *        kobj;
const char *            fmt;
                        ...;
```

Arguments

drv

requesting device driver

kobj

kobject to add below this driver

fmt

format string that names the kobject

...

variable arguments

Description

You really don't want to do this, this is only here due to one looney iseries driver, go poke those developers if you are annoyed about this...

Name

get_driver — increment driver reference count.

Synopsis

```
struct device_driver * get_driver (drv);

struct device_driver *  drv;
```

Arguments

drv

driver.

Name

put_driver — decrement driver's refcount.

Synopsis

```
void put_driver (drv);

struct device_driver *  drv;
```

Arguments

drv

driver.

Name

driver_register — register driver with bus

Synopsis

```
int driver_register (drv);

struct device_driver *  drv;
```

Arguments

drv

driver to register

Description

We pass off most of the work to the `bus_add_driver` call, since most of the things we have to do deal with the bus structures.

Name

driver_unregister — remove driver from system.

Synopsis

```
void driver_unregister (drv);

struct device_driver *  drv;
```

Arguments

drv

driver.

Description

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

Name

driver_find — locate driver on a bus by its name.

Synopsis

```
struct device_driver * driver_find (name,
                                    bus);

const char *       name;
struct bus_type *  bus;
```

Arguments

name

name of the driver.

bus

bus to scan for the driver.

Description

Call `kset_find_obj` to iterate over list of drivers on a bus to find driver by name. Return driver if found.

Note that kset_find_obj increments driver's reference count.

Name

dev_driver_string — Return a device's driver name, if at all possible

Synopsis

```
const char * dev_driver_string (dev);

const struct device *  dev;
```

Arguments

dev

struct device to get the name of

Description

Will return the device's driver's name if it is bound to a device. If the device is not bound to a device, it will return the name of the bus it is attached to. If it is not attached to a bus either, an empty string will be returned.

Name

device_create_file — create sysfs attribute file for device.

Synopsis

```
int device_create_file (dev,
                        attr);

struct device *            dev;
struct device_attribute *  attr;
```

Arguments

dev

device.

attr

device attribute descriptor.

Name

device_remove_file — remove sysfs attribute file.

Synopsis

```
void device_remove_file (dev,
                         attr);

struct device *            dev;
struct device_attribute *  attr;
```

Arguments

dev

device.

attr

device attribute descriptor.

Name

device_create_bin_file — create sysfs binary attribute file for device.

Synopsis

```
int device_create_bin_file (dev,
                            attr);

struct device *         dev;
struct bin_attribute *  attr;
```

Arguments

dev

device.

attr

device binary attribute descriptor.

Name

device_remove_bin_file — remove sysfs binary attribute file

Synopsis

```
void device_remove_bin_file (dev,
                             attr);

struct device *         dev;
struct bin_attribute *  attr;
```

Arguments

dev

device.

attr

device binary attribute descriptor.

Name

device_schedule_callback_owner — helper to schedule a callback for a device

Synopsis

```
int device_schedule_callback_owner (dev,
                                    func,
                                    owner);

struct device *  dev;
void (*          func(struct device *);
struct module *  owner;
```

Arguments

dev

device.

func

callback function to invoke later.

owner

module owning the callback routine

Description

Attribute methods must not unregister themselves or their parent device (which would amount to the same thing). Attempts to do so will deadlock, since unregistration is mutually exclusive with driver callbacks.

Instead methods can call this routine, which will attempt to allocate and schedule a workqueue request to call back *func* with *dev* as its argument in the workqueue's process context. *dev* will be pinned until *func* returns.

This routine is usually called via the inline device_schedule_callback, which automatically sets *owner* to THIS_MODULE.

Returns 0 if the request was submitted, -ENOMEM if storage could not be allocated, -ENODEV if a reference to *owner* isn't available.

NOTE

This routine won't work if CONFIG_SYSFS isn't set! It uses an underlying sysfs routine (since it is intended for use by attribute methods), and if sysfs isn't available you'll get nothing but -ENOSYS.

Name

device_initialize — init device structure.

Synopsis

```
void device_initialize (dev);

struct device *  dev;
```

Arguments

dev

device.

Description

This prepares the device for use by other layers by initializing its fields. It is the first half of device_register, if called by that function, though it can also be called separately, so one may use *dev*'s fields. In particular, get_device/put_device may be used for reference counting of *dev* after calling this function.

NOTE

Use `put_device` to give up your reference instead of freeing *`dev`* directly once you have called this function.

Name

dev_set_name — set a device name

Synopsis

```
int dev_set_name (dev,
                  fmt,
                  ...);

struct device *  dev;
const char *     fmt;
                 ...;
```

Arguments

`dev`

device

`fmt`

format string for the device's name

`...`

variable arguments

Name

device_add — add device to device hierarchy.

Synopsis

```
int device_add (dev);

struct device *  dev;
```

Arguments

dev

device.

Description

This is part 2 of `device_register`, though may be called separately _iff_ `device_initialize` has been called separately.

This adds *dev* to the kobject hierarchy via `kobject_add`, adds it to the global and sibling lists for the device, then adds it to the other relevant subsystems of the driver model.

NOTE

Never directly free *dev* after calling this function, even if it returned an error! Always use `put_device` to give up your reference instead.

Name

device_register — register a device with the system.

Synopsis

```
int device_register (dev);

struct device *  dev;
```

Arguments

dev

pointer to the device structure

Description

This happens in two clean steps - initialize the device and add it to the system. The two steps can be called separately, but this is the easiest and most common. I.e. you should only call the two helpers separately if have a clearly defined need to use and refcount the device before it is added to the hierarchy.

NOTE

Never directly free *dev* after calling this function, even if it returned an error! Always use `put_device` to give up the reference initialized in this function instead.

Name

get_device — increment reference count for device.

Synopsis

```
struct device * get_device (dev);

struct device *  dev;
```

Arguments

dev

device.

Description

This simply forwards the call to `kobject_get`, though we do take care to provide for the case that we get a NULL pointer passed in.

Name

put_device — decrement reference count.

Synopsis

```
void put_device (dev);

struct device *  dev;
```

Arguments

dev

device in question.

Name

device_del — delete device from system.

Synopsis

```
void device_del (dev);

struct device *  dev;
```

Arguments

dev

device.

Description

This is the first part of the device unregistration sequence. This removes the device from the lists we control from here, has it removed from the other driver model subsystems it was added to in `device_add`, and removes it from the kobject hierarchy.

NOTE

this should be called manually _iff_ `device_add` was also called manually.

Name

device_unregister — unregister device from system.

Synopsis

```
void device_unregister (dev);

struct device *  dev;
```

Arguments

dev

device going away.

Description

We do this in two parts, like we do `device_register`. First, we remove it from all the subsystems with `device_del`, then we decrement the reference count via `put_device`. If that is the final reference count, the device will be cleaned up via `device_release` above. Otherwise, the structure will stick around until the final reference to the device is dropped.

Name

device_for_each_child — device child iterator.

Synopsis

```
int device_for_each_child (parent,
                           data,
                           fn);

struct device *  parent;
void *           data;
int (*           fn(struct device *dev, void *data);
```

Arguments

parent

parent struct device.

data

data for the callback.

fn

function to be called for each device.

Description

Iterate over *parent*'s child devices, and call *fn* for each, passing it *data*.

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

Name

device_find_child — device iterator for locating a particular device.

Synopsis

```
struct device * device_find_child (parent,
                                   data,
                                   match);

struct device *  parent;
void *           data;
int (*           match(struct device *dev, void *data);
```

Arguments

parent

parent struct device

data

Data to pass to match function

match

Callback function to check device

Description

This is similar to the `device_for_each_child` function above, but it returns a reference to a device that is 'found' for later use, as determined by the *match* callback.

The callback should return 0 if the device doesn't match and non-zero if it does. If the callback returns non-zero and a reference to the current device can be obtained, this function will return to the caller and not iterate over any more devices.

Name

__root_device_register — allocate and register a root device

Synopsis

```
struct device * __root_device_register (name,
                                        owner);

const char *     name;
struct module *  owner;
```

Arguments

name

root device name

owner

owner module of the root device, usually THIS_MODULE

Description

This function allocates a root device and registers it using `device_register`. In order to free the

returned device, use `root_device_unregister`.

Root devices are dummy devices which allow other devices to be grouped under /sys/devices. Use this function to allocate a root device and then use it as the parent of any device which should appear under /sys/devices/{name}

The /sys/devices/{name} directory will also contain a 'module' symlink which points to the *owner* directory in sysfs.

Note

You probably want to use `root_device_register`.

Name

root_device_unregister — unregister and free a root device

Synopsis

```
void root_device_unregister (dev);

struct device *  dev;
```

Arguments

dev

device going away

Description

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

Name

device_create_vargs — creates a device and registers it with sysfs

Synopsis

```
struct device * device_create_vargs (class,
                                     parent,
                                     devt,
                                     drvdata,
                                     fmt,
                                     args);
```

```
struct class *   class;
struct device *  parent;
dev_t            devt;
void *           drvdata;
const char *     fmt;
va_list          args;
```

Arguments

class

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

parent

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

devt

the dev_t for the char device to be added

drvdata

the data to be added to the device for callbacks

fmt

string for the device's name

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.

Note

the struct class passed to this function must have previously been created with a call to `class_create`.

Name

device_create — creates a device and registers it with sysfs

Synopsis

```
struct device * device_create (class,
                               parent,
                               devt,
                               drvdata,
                               fmt,
                               ...);

struct class *   class;
struct device *  parent;
dev_t            devt;
void *           drvdata;
const char *     fmt;
                 ...;
```

Arguments

class

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

parent

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

devt

the dev_t for the char device to be added

drvdata

the data to be added to the device for callbacks

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.

Note

the struct class passed to this function must have previously been created with a call to `class_create`.

Name

device_destroy — removes a device that was created with `device_create`

Synopsis

```
void device_destroy (class,
                     devt);

struct class *  class;
dev_t           devt;
```

Arguments

class

pointer to the struct class that this device was registered with

devt

the dev_t of the device that was previously registered

Description

This call unregisters and cleans up a device that was created with a call to `device_create`.

Name

device_rename — renames a device

Synopsis

```
int device_rename (dev,
                   new_name);

struct device *  dev;
char *           new_name;
```

Arguments

dev

the pointer to the struct device to be renamed

new_name

the new name of the device

Description

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

Name

device_move — moves a device to a new parent

Synopsis

```
int device_move (dev,
                 new_parent,
                 dpm_order);

struct device *  dev;
struct device *  new_parent;
enum dpm_order   dpm_order;
```

Arguments

dev

the pointer to the struct device to be moved

new_parent

the new parent of the device (can by NULL)

dpm_order

how to reorder the dpm_list

Name

__class_create — create a struct class structure

Synopsis

```
struct class * __class_create (owner,
                               name,
                               key);

struct module *          owner;
const char *             name;
struct lock_class_key *  key;
```

Arguments

owner

pointer to the module that is to “own” this struct class

name

pointer to a string for the name of this class.

key

the lock_class_key for this class; used by mutex lock debugging

Description

This is used to create a struct class pointer that can then be used in calls to `device_create`.

Note, the pointer created here is to be destroyed when finished by making a call to `class_destroy`.

Name

class_destroy — destroys a struct class structure

Synopsis

```
void class_destroy (cls);

struct class *  cls;
```

Arguments

cls

pointer to the struct class that is to be destroyed

Description

Note, the pointer to be destroyed must have been created with a call to `class_create`.

Name

class_dev_iter_init — initialize class device iterator

Synopsis

```
void class_dev_iter_init (iter,
                          class,
                          start,
                          type);

struct class_dev_iter *     iter;
struct class *              class;
struct device *             start;
const struct device_type *  type;
```

Arguments

iter

class iterator to initialize

class

the class we wanna iterate over

start

the device to start iterating from, if any

type

device_type of the devices to iterate over, NULL for all

Description

Initialize class iterator *iter* such that it iterates over devices of *class*. If *start* is set, the list iteration will start there, otherwise if it is NULL, the iteration starts at the beginning of the list.

Name

class_dev_iter_next — iterate to the next device

Synopsis

```
struct device * class_dev_iter_next (iter);

struct class_dev_iter *  iter;
```

Arguments

iter

class iterator to proceed

Description

Proceed *iter* to the next device and return it. Returns NULL if iteration is complete.

The returned device is referenced and won't be released till iterator is proceed to the next device or exited. The caller is free to do whatever it wants to do with the device including calling back into class code.

Name

class_dev_iter_exit — finish iteration

Synopsis

```
void class_dev_iter_exit (iter);

struct class_dev_iter *  iter;
```

Arguments

iter

class iterator to finish

Description

Finish an iteration. Always call this function after iteration is complete whether the iteration ran till the end or not.

Name

class_for_each_device — device iterator

Synopsis

```
int class_for_each_device (class,
                           start,
                           data,
                           fn);

struct class *   class;
struct device *  start;
void *           data;
int (*           fn(struct device *, void *);
```

Arguments

class

the class we're iterating

start

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

data

data for the callback

fn

function to be called for each device

Description

Iterate over *class*'s list of devices, and call *fn* for each, passing it *data*. If *start* is set, the list iteration will start there, otherwise if it is NULL, the iteration starts at the beginning of the list.

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

fn is allowed to do anything including calling back into class code. There's no locking restriction.

Name

class_find_device — device iterator for locating a particular device

Synopsis

```
struct device * class_find_device (class,
                                   start,
                                   data,
                                   match);

struct class *   class;
struct device *  start;
void *           data;
int (*           match(struct device *, void *);
```

Arguments

class

the class we're iterating

start

Device to begin with

data

data for the match function

match

function to check device

Description

This is similar to the `class_for_each_dev` function above, but it returns a reference to a device that is 'found' for later use, as determined by the *match* callback.

The callback should return 0 if the device doesn't match and non-zero if it does. If the callback returns non-zero, this function will return to the caller and not iterate over any more devices.

Note, you will need to drop the reference with `put_device` after use.

fn is allowed to do anything including calling back into class code. There's no locking restriction.

Name

class_compat_register — register a compatibility class

Synopsis

```
struct class_compat * class_compat_register (name);

const char *  name;
```

Arguments

name

the name of the class

Description

Compatibility class are meant as a temporary user-space compatibility workaround when converting a family of class devices to a bus devices.

Name

class_compat_unregister — unregister a compatibility class

Synopsis

```
void class_compat_unregister (cls);

struct class_compat *  cls;
```

Arguments

cls

the class to unregister

Name

class_compat_create_link — create a compatibility class device link to a bus device

Synopsis

```
int class_compat_create_link (cls,
                              dev,
                              device_link);

struct class_compat *  cls;
struct device *        dev;
struct device *        device_link;
```

Arguments

cls

the compatibility class

dev

the target bus device

device_link

an optional device to which a “device” link should be created

---

Name

class_compat_remove_link — remove a compatibility class device link to a bus device

Synopsis

```
void class_compat_remove_link (cls,
                               dev,
                               device_link);

struct class_compat *  cls;
struct device *        dev;
struct device *        device_link;
```

Arguments

cls

the compatibility class

dev

the target bus device

device_link

an optional device to which a “device” link was previously created

---

Name

request_firmware — send firmware request and wait for it

Synopsis

```
int request_firmware (firmware_p,
                      name,
```

device);

```
const struct firmware **  firmware_p;
const char *              name;
struct device *           device;
```

Arguments

firmware_p

pointer to firmware image

name

name of firmware file

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.

Name

release_firmware — release the resource associated with a firmware image

Synopsis

void **release_firmware** (*fw*);

```
const struct firmware *  fw;
```

Arguments

fw

firmware resource to release

Name

request_firmware_nowait —

Synopsis

```
int request_firmware_nowait (module,
                             uevent,
                             name,
                             device,
                             context,
                             cont);

struct module *  module;
int              uevent;
const char *     name;
struct device *  device;
void *           context;
void (*          cont(const struct firmware *fw, void *context);
```

Arguments

module

module requesting the firmware

uevent

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

name

name of firmware file

device

device for which firmware is being loaded

context

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

cont

function will be called asynchronously when the firmware request is over.

Description

Asynchronous variant of request_firmware for user contexts where it is not possible to sleep for long time. It can't be called in atomic contexts.

Name

transport_class_register — register an initial transport class

Synopsis

```
int transport_class_register (tclass);

struct transport_class *  tclass;
```

Arguments

tclass

a pointer to the transport class structure to be initialised

Description

The transport class contains an embedded class which is used to identify it. The caller should initialise this structure with zeros and then generic class must have been initialised with the actual transport class unique name. There's a macro DECLARE_TRANSPORT_CLASS to do this (declared classes still must be registered).

Returns 0 on success or error on failure.

Name

transport_class_unregister — unregister a previously registered class

Synopsis

```
void transport_class_unregister (tclass);

struct transport_class *  tclass;
```

Arguments

tclass

The transport class to unregister

Description

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

Name

anon_transport_class_register — register an anonymous class

Synopsis

```
int anon_transport_class_register (atc);

struct anon_transport_class *  atc;
```

Arguments

atc

The anon transport class to register

Description

The anonymous transport class contains both a transport class and a container. The idea of an anonymous class is that it never actually has any device attributes associated with it (and thus saves on container storage). So it can only be used for triggering events. Use prezero and then use DECLARE_ANON_TRANSPORT_CLASS to initialise the anon transport class storage.

Name

anon_transport_class_unregister — unregister an anon class

Synopsis

```
void anon_transport_class_unregister (atc);

struct anon_transport_class *  atc;
```

Arguments

atc

Pointer to the anon transport class to unregister

Description

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

Name

transport_setup_device — declare a new dev for transport class association but don't make it visible yet.

Synopsis

```
void transport_setup_device (dev);

struct device *  dev;
```

Arguments

dev

the generic device representing the entity being added

Description

Usually, dev represents some component in the HBA system (either the HBA itself or a device remote across the HBA bus). This routine is simply a trigger point to see if any set of transport classes wishes to associate with the added device. This allocates storage for the class device and initialises it, but does not yet add it to the system or add attributes to it (you do this with transport_add_device). If you have no need for a separate setup and add operations, use transport_register_device (see transport_class.h).

Name

transport_add_device — declare a new dev for transport class association

Synopsis

```
void transport_add_device (dev);

struct device *  dev;
```

Arguments

dev

the generic device representing the entity being added

Description

Usually, dev represents some component in the HBA system (either the HBA itself or a device remote across the HBA bus). This routine is simply a trigger point used to add the device to the system and register attributes for it.

Name

transport_configure_device — configure an already set up device

Synopsis

```
void transport_configure_device (dev);

struct device *  dev;
```

Arguments

dev

generic device representing device to be configured

Description

The idea of configure is simply to provide a point within the setup process to allow the transport class to extract information from a device after it has been setup. This is used in SCSI because we have to have a setup device to begin using the HBA, but after we send the initial inquiry, we use configure to extract the device parameters. The device need not have been added to be configured.

Name

transport_remove_device — remove the visibility of a device

Synopsis

```
void transport_remove_device (dev);

struct device *  dev;
```

Arguments

dev

generic device to remove

Description

This call removes the visibility of the device (to the user from sysfs), but does not destroy it. To eliminate a device entirely you must also call transport_destroy_device. If you don't need to do remove and destroy as separate operations, use `transport_unregister_device` (see transport_class.h) which

will perform both calls for you.

Name

transport_destroy_device — destroy a removed device

Synopsis

```
void transport_destroy_device (dev);

struct device *  dev;
```

Arguments

dev

device to eliminate from the transport class.

Description

This call triggers the elimination of storage associated with the transport classdev. Note: all it really does is relinquish a reference to the classdev. The memory will not be freed until the last reference goes to zero. Note also that the classdev retains a reference count on dev, so dev too will remain for as long as the transport class device remains around.

Name

sysdev_driver_register — Register auxillary driver

Synopsis

```
int sysdev_driver_register (cls,
                            drv);

struct sysdev_class *   cls;
struct sysdev_driver *  drv;
```

Arguments

cls

Device class driver belongs to.

drv

Driver.

Description

drv is inserted into *cls*->drivers to be called on each operation on devices of that class. The refcount of *cls* is incremented.

Name

sysdev_driver_unregister — Remove an auxillary driver.

Synopsis

```
void sysdev_driver_unregister (cls,
                               drv);

struct sysdev_class *   cls;
struct sysdev_driver *  drv;
```

Arguments

cls

Class driver belongs to.

drv

Driver.

Name

sysdev_register — add a system device to the tree

Synopsis

```
int sysdev_register (sysdev);

struct sys_device *  sysdev;
```

Arguments

sysdev

device in question

Name

sysdev_suspend — Suspend all system devices.

Synopsis

```
int sysdev_suspend (state);

pm_message_t  state;
```

Arguments

state

Power state to enter.

Description

We perform an almost identical operation as `sysdev_shutdown` above, though calling `->suspend` instead. Interrupts are disabled when this called. Devices are responsible for both saving state and quiescing or powering down the device.

This is only called by the device PM core, so we let them handle all synchronization.

Name

sysdev_resume — Bring system devices back to life.

Synopsis

```
int sysdev_resume (void);

 void;
```

Arguments

void

no arguments

Description

Similar to `sysdev_suspend`, but we iterate the list forwards to guarantee that parent devices are resumed before their children.

Note

Interrupts are disabled when called.

Name

platform_get_resource — get a resource for a device

Synopsis

```
struct resource * platform_get_resource (dev,
                                         type,
                                         num);

struct platform_device *  dev;
unsigned int              type;
unsigned int              num;
```

Arguments

dev

platform device

type

resource type

num

resource index

Name

platform_get_irq — get an IRQ for a device

Synopsis

```
int platform_get_irq (dev,
                      num);

struct platform_device *  dev;
unsigned int              num;
```

Arguments

dev

platform device

num

IRQ number index

Name

platform_get_resource_byname — get a resource for a device by name

Synopsis

```
struct resource * platform_get_resource_byname (dev,
                                                type,
                                                name);

struct platform_device *  dev;
unsigned int              type;
const char *              name;
```

Arguments

dev

platform device

type

resource type

name

resource name

Name

platform_get_irq_byname — get an IRQ for a device

Synopsis

```
int platform_get_irq_byname (dev,
                             name);

struct platform_device *  dev;
const char *              name;
```

Arguments

dev

platform device

name

IRQ name

Name

platform_add_devices — add a numbers of platform devices

Synopsis

```
int platform_add_devices (devs,
                          num);

struct platform_device **  devs;
int                        num;
```

Arguments

devs

array of platform devices to add

num

number of platform devices in array

Name

platform_device_put —

Synopsis

```
void platform_device_put (pdev);

struct platform_device *  pdev;
```

Arguments

pdev

platform device to free

Description

Free all memory associated with a platform device. This function must _only_ be externally called in error cases. All other usage is a bug.

Name

platform_device_alloc —

Synopsis

```
struct platform_device * platform_device_alloc (name,
                                                id);

const char *  name;
int           id;
```

Arguments

name

base name of the device we're adding

id

instance id

Description

Create a platform device object which can have other objects attached to it, and which will have attached objects freed when it is released.

Name

platform_device_add_resources —

Synopsis

```
int platform_device_add_resources (pdev,
                                   res,
                                   num);
```

```
struct platform_device *  pdev;
struct resource *         res;
unsigned int              num;
```

Arguments

pdev

platform device allocated by platform_device_alloc to add resources to

res

set of resources that needs to be allocated for the device

num

number of resources

Description

Add a copy of the resources to the platform device. The memory associated with the resources will be freed when the platform device is released.

Name

platform_device_add_data —

Synopsis

```
int platform_device_add_data (pdev,
                              data,
                              size);

struct platform_device *  pdev;
const void *              data;
size_t                    size;
```

Arguments

pdev

platform device allocated by platform_device_alloc to add resources to

data

platform specific data for this platform device

size

size of platform specific data

Description

Add a copy of platform specific data to the platform device's platform_data pointer. The memory associated with the platform data will be freed when the platform device is released.

Name

platform_device_add — add a platform device to device hierarchy

Synopsis

```
int platform_device_add (pdev);

struct platform_device * pdev;
```

Arguments

pdev

platform device we're adding

Description

This is part 2 of `platform_device_register`, though may be called separately _iff_ pdev was allocated by `platform_device_alloc`.

Name

platform_device_del — remove a platform-level device

Synopsis

```
void platform_device_del (pdev);

struct platform_device * pdev;
```

Arguments

pdev

platform device we're removing

Description

Note that this function will also release all memory- and port-based resources owned by the device (*dev*->resource). This function must _only_ be externally called in error cases. All other usage is a bug.

Name

platform_device_register — add a platform-level device

Synopsis

```
int platform_device_register (pdev);

struct platform_device *  pdev;
```

Arguments

pdev

platform device we're adding

Name

platform_device_unregister — unregister a platform-level device

Synopsis

```
void platform_device_unregister (pdev);

struct platform_device *  pdev;
```

Arguments

pdev

platform device we're unregistering

Description

Unregistration is done in 2 steps. First we release all resources and remove it from the subsystem, then we drop reference count by calling `platform_device_put`.

Name

platform_device_register_simple —

Synopsis

```
struct platform_device * platform_device_register_simple (name,
                                                          id,
                                                          res,
                                                          num);

const char *       name;
int                id;
struct resource *  res;
unsigned int       num;
```

Arguments

name

base name of the device we're adding

id

instance id

res

set of resources that needs to be allocated for the device

num

number of resources

Description

This function creates a simple platform device that requires minimal resource and memory management. Canned release function freeing memory allocated for the device allows drivers using such devices to be unloaded without waiting for the last reference to the device to be dropped.

This interface is primarily intended for use with legacy drivers which probe hardware directly. Because such drivers create sysfs device nodes themselves, rather than letting system infrastructure handle such device enumeration tasks, they don't fully conform to the Linux driver model. In particular, when such drivers are built as modules, they can't be “hotplugged”.

Name

platform_driver_register —

Synopsis

```
int platform_driver_register (drv);

struct platform_driver *  drv;
```

Arguments

drv

platform driver structure

Name

platform_driver_unregister —

Synopsis

```
void platform_driver_unregister (drv);

struct platform_driver *  drv;
```

Arguments

drv

platform driver structure

Name

platform_driver_probe — register driver for non-hotpluggable device

Synopsis

```
int __init_or_module platform_driver_probe (drv,
                                            probe);

struct platform_driver *  drv;
int (*                    probe(struct platform_device *);
```

Arguments

drv

platform driver structure

probe

the driver probe routine, probably from an __init section

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.

Returns zero if the driver registered and bound to a device, else returns a negative error code and with the driver not registered.

Name

bus_for_each_dev — device iterator.

Synopsis

```
int bus_for_each_dev (bus,
                      start,
                      data,
                      fn);

struct bus_type *  bus;
struct device *    start;
void *             data;
int (*             fn(struct device *, void *);
```

Arguments

bus

bus type.

start

device to start iterating from.

data

data for the callback.

fn

function to be called for each device.

Description

Iterate over *bus*'s list of devices, and call *fn* for each, passing it *data*. If *start* is not NULL, we use that device to begin iterating from.

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

NOTE

The device that returns a non-zero value is not retained in any way, nor is its refcount incremented. If the caller needs to retain this data, it should do so, and increment the reference count in the supplied callback.

Name

bus_find_device — device iterator for locating a particular device.

Synopsis

```
struct device * bus_find_device (bus,
                                 start,
                                 data,
                                 match);

struct bus_type *  bus;
struct device *    start;
void *             data;
int (*             match(struct device *dev, void *data);
```

Arguments

bus

bus type

start

Device to begin with

data

Data to pass to match function

match

Callback function to check device

Description

This is similar to the `bus_for_each_dev` function above, but it returns a reference to a device that is 'found' for later use, as determined by the *`match`* callback.

The callback should return 0 if the device doesn't match and non-zero if it does. If the callback returns non-zero, this function will return to the caller and not iterate over any more devices.

Name

bus_find_device_by_name — device iterator for locating a particular device of a specific name

Synopsis

```
struct device * bus_find_device_by_name (bus,
                                         start,
                                         name);

struct bus_type *  bus;
struct device *    start;
const char *       name;
```

Arguments

`bus`

bus type

`start`

Device to begin with

`name`

name of the device to match

Description

This is similar to the `bus_find_device` function above, but it handles searching by a name automatically, no need to write another strcmp matching function.

Name

bus_for_each_drv — driver iterator

Synopsis

```
int bus_for_each_drv (bus,
                      start,
                      data,
                      fn);

struct bus_type *       bus;
struct device_driver *  start;
void *                  data;
int (*                  fn(struct device_driver *, void *);
```

Arguments

bus

bus we're dealing with.

start

driver to start iterating on.

data

data to pass to the callback.

fn

function to call for each driver.

Description

This is nearly identical to the device iterator above. We iterate over each driver that belongs to *bus*, and call *fn* for each. If *fn* returns anything but 0, we break out and return it. If *start* is not NULL, we use it as the head of the list.

NOTE

we don't return the driver that returns a non-zero value, nor do we leave the reference count incremented for that driver. If the caller needs to know that info, it must set it in the callback. It must also be sure to increment the refcount so it doesn't disappear before returning to the caller.

Name

bus_rescan_devices — rescan devices on the bus for possible drivers

Synopsis

```
int bus_rescan_devices (bus);

struct bus_type *  bus;
```

Arguments

bus

the bus to scan.

Description

This function will look for devices on the bus with no driver attached and rescan it against existing drivers to see if it matches any by calling `device_attach` for the unbound devices.

Name

device_reprobe — remove driver for a device and probe for a new driver

Synopsis

```
int device_reprobe (dev);

struct device *  dev;
```

Arguments

dev

the device to reprobe

Description

This function detaches the attached driver (if any) for the given device and restarts the driver probing process. It is intended to use if probing criteria changed during a devices lifetime and driver attachment should change accordingly.

Name

bus_register — register a bus with the system.

Synopsis

```
int bus_register (bus);

struct bus_type *  bus;
```

Arguments

bus

bus.

Description

Once we have that, we registered the bus with the kobject infrastructure, then register the children subsystems it has: the devices and drivers that belong to the bus.

Name

bus_unregister — remove a bus from the system

Synopsis

```
void bus_unregister (bus);

struct bus_type *  bus;
```

Arguments

bus

bus.

Description

Unregister the child subsystems and the bus itself. Finally, we call `bus_put` to release the refcount

Device Drivers Power Management

Name

dpm_resume_noirq — Execute “early resume” callbacks for non-sysdev devices.

Synopsis

```
void dpm_resume_noirq (state);

pm_message_t  state;
```

Arguments

state

PM transition of the system being carried out.

Description

Call the “noirq” resume handlers for all devices marked as DPM_OFF_IRQ and enable device drivers to receive interrupts.

Name

dpm_resume_end — Execute “resume” callbacks and complete system transition.

Synopsis

```
void dpm_resume_end (state);

pm_message_t  state;
```

Arguments

state

PM transition of the system being carried out.

Description

Execute “resume” callbacks for all devices and complete the PM transition of the system.

Name

dpm_suspend_noirq — Execute “late suspend” callbacks for non-sysdev devices.

Synopsis

```
int dpm_suspend_noirq (state);
```

```
pm_message_t state;
```

Arguments

state

PM transition of the system being carried out.

Description

Prevent device drivers from receiving interrupts and call the “noirq” suspend handlers for all non-sysdev devices.

Name

dpm_suspend_start — Prepare devices for PM transition and suspend them.

Synopsis

```
int dpm_suspend_start (state);

pm_message_t state;
```

Arguments

state

PM transition of the system being carried out.

Description

Prepare all non-sysdev devices for system PM transition and execute “suspend” callbacks for them.

Device Drivers ACPI Support

Name

acpi_bus_register_driver — register a driver with the ACPI bus

Synopsis

```
int acpi_bus_register_driver (driver);

struct acpi_driver * driver;
```

Arguments

driver

driver being registered

Description

Registers a driver with the ACPI bus. Searches the namespace for all devices that match the driver's criteria and binds. Returns zero for success or a negative error status for failure.

Name

acpi_bus_unregister_driver — unregisters a driver with the APIC bus

Synopsis

```
void acpi_bus_unregister_driver (driver);

struct acpi_driver *  driver;
```

Arguments

driver

driver to unregister

Description

Unregisters a driver with the ACPI bus. Searches the namespace for all devices that match the driver's criteria and unbinds.

Name

acpi_bus_driver_init — add a device to a driver

Synopsis

```
int acpi_bus_driver_init (device,
                          driver);

struct acpi_device *  device;
struct acpi_driver *  driver;
```

Arguments

device

the device to add and initialize

driver

driver for the device

Description

Used to initialize a device via its device driver. Called whenever a driver is bound to a device. Invokes the driver's `add` ops.

Device drivers PnP support

Name

pnp_register_protocol — adds a pnp protocol to the pnp layer

Synopsis

```
int pnp_register_protocol (protocol);

struct pnp_protocol * protocol;
```

Arguments

protocol

pointer to the corresponding pnp_protocol structure

Ex protocols

ISAPNP, PNPBIOS, etc

Name

pnp_unregister_protocol — removes a pnp protocol from the pnp layer

Synopsis

```
void pnp_unregister_protocol (protocol);
```

```
struct pnp_protocol *  protocol;
```

Arguments

protocol

pointer to the corresponding pnp_protocol structure

Name

pnp_request_card_device — Searches for a PnP device under the specified card

Synopsis

```
struct pnp_dev * pnp_request_card_device (clink,
                                          id,
                                          from);

struct pnp_card_link *  clink;
const char *            id;
struct pnp_dev *        from;
```

Arguments

clink

pointer to the card link, cannot be NULL

id

pointer to a PnP ID structure that explains the rules for finding the device

from

Starting place to search from. If NULL it will start from the begining.

Name

pnp_release_card_device — call this when the driver no longer needs the device

Synopsis

```
void pnp_release_card_device (dev);

struct pnp_dev *  dev;
```

Arguments

dev

pointer to the PnP device stucture

Name

pnp_register_card_driver — registers a PnP card driver with the PnP Layer

Synopsis

```
int pnp_register_card_driver (drv);

struct pnp_card_driver *  drv;
```

Arguments

drv

pointer to the driver to register

Name

pnp_unregister_card_driver — unregisters a PnP card driver from the PnP Layer

Synopsis

```
void pnp_unregister_card_driver (drv);

struct pnp_card_driver *  drv;
```

Arguments

drv

pointer to the driver to unregister

Name

pnp_add_id — adds an EISA id to the specified device

Synopsis

```
struct pnp_id * pnp_add_id (dev,
                            id);

struct pnp_dev *  dev;
char *            id;
```

Arguments

dev

pointer to the desired device

id

pointer to an EISA id string

Name

pnp_start_dev — low-level start of the PnP device

Synopsis

```
int pnp_start_dev (dev);

struct pnp_dev *  dev;
```

Arguments

dev

pointer to the desired device

Description

assumes that resources have already been allocated

Name

pnp_stop_dev — low-level disable of the PnP device

Synopsis

```
int pnp_stop_dev (dev);
```

```
struct pnp_dev * dev;
```

Arguments

dev

pointer to the desired device

Description

does not free resources

Name

pnp_activate_dev — activates a PnP device for use

Synopsis

```
int pnp_activate_dev (dev);

struct pnp_dev * dev;
```

Arguments

dev

pointer to the desired device

Description

does not validate or set resources so be careful.

Name

pnp_disable_dev — disables device

Synopsis

```
int pnp_disable_dev (dev);

struct pnp_dev * dev;
```

Arguments

dev

pointer to the desired device

Description

inform the correct pnp protocol so that resources can be used by other devices

---

Name

pnp_is_active — Determines if a device is active based on its current resources

Synopsis

```
int pnp_is_active (dev);

struct pnp_dev *  dev;
```

Arguments

dev

pointer to the desired PnP device

Userspace IO devices

Name

uio_event_notify — trigger an interrupt event

Synopsis

```
void uio_event_notify (info);

struct uio_info *  info;
```

Arguments

info

UIO device capabilities

Name

__uio_register_device — register a new userspace IO device

Synopsis

```
int __uio_register_device (owner,
                           parent,
                           info);

struct module *    owner;
struct device *    parent;
struct uio_info *  info;
```

Arguments

owner

module that creates the new device

parent

parent device

info

UIO device capabilities

Description

returns zero on success or a negative error code.

Name

uio_unregister_device — unregister a industrial IO device

Synopsis

```
void uio_unregister_device (info);

struct uio_info *  info;
```

Arguments

info

UIO device capabilities

Name

struct uio_mem — description of a UIO memory region

Synopsis

```
struct uio_mem {
  const char * name;
  unsigned long addr;
  unsigned long 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

size

size of IO

memtype

type of memory addr points to

internal_addr

ioremap-ped version of addr, for driver internal use

map

for use by the UIO core only.

Name

struct uio_port — description of a UIO port region

Synopsis

```
struct uio_port {
  const char * name;
```

```
  unsigned long start;
  unsigned long size;
  int porttype;
  struct uio_portio * portio;
};
```

Members

name

name of the port region for identification

start

start of port region

size

size of port region

porttype

type of port (see UIO_PORT_* below)

portio

for use by the UIO core only.

Name

struct uio_info — UIO device capabilities

Synopsis

```
struct uio_info {
  struct uio_device * uio_dev;
  const char * name;
  const char * version;
  struct uio_mem mem[MAX_UIO_MAPS];
  struct uio_port port[MAX_UIO_PORT_REGIONS];
  long irq;
  unsigned long irq_flags;
  void * priv;
  irqreturn_t (* handler) (int irq, struct uio_info *dev_info);
  int (* mmap) (struct uio_info *info, struct vm_area_struct *vma);
  int (* open) (struct uio_info *info, struct inode *inode);
  int (* release) (struct uio_info *info, struct inode *inode);
  int (* irqcontrol) (struct uio_info *info, s32 irq_on);
};
```

Members

uio_dev

the UIO device this info belongs to

name

device name

version

device driver version

mem[MAX_UIO_MAPS]

list of mappable memory regions, size==0 for end of list

port[MAX_UIO_PORT_REGIONS]

list of port regions, size==0 for end of list

irq

interrupt number or UIO_IRQ_CUSTOM

irq_flags

flags for `request_irq`

priv

optional private data

handler

the device's irq handler

mmap

mmap operation for this uio device

open

open operation for this uio device

release

release operation for this uio device

irqcontrol

disable/enable irqs when 0/1 is written to /dev/uioX

Chapter 3. Parallel Port Devices

Table of Contents

Name

parport_yield — relinquish a parallel port temporarily

Synopsis

```
int parport_yield (dev);

struct pardevice *  dev;
```

Arguments

dev

a device on the parallel port

Description

This function relinquishes the port if it would be helpful to other drivers to do so. Afterwards it tries to

reclaim the port using `parport_claim`, and the return value is the same as for `parport_claim`. If it fails, the port is left unclaimed and it is the driver's responsibility to reclaim the port.

The `parport_yield` and `parport_yield_blocking` functions are for marking points in the driver at which other drivers may claim the port and use their devices. Yielding the port is similar to releasing it and reclaiming it, but is more efficient because no action is taken if there are no other devices needing the port. In fact, nothing is done even if there are other devices waiting but the current device is still within its “timeslice”. The default timeslice is half a second, but it can be adjusted via the /proc interface.

Name

parport_yield_blocking — relinquish a parallel port temporarily

Synopsis

```
int parport_yield_blocking (dev);

struct pardevice *  dev;
```

Arguments

dev

a device on the parallel port

Description

This function relinquishes the port if it would be helpful to other drivers to do so. Afterwards it tries to reclaim the port using `parport_claim_or_block`, and the return value is the same as for `parport_claim_or_block`.

Name

parport_wait_event — wait for an event on a parallel port

Synopsis

```
int parport_wait_event (port,
                        timeout);

struct parport *  port;
signed long       timeout;
```

Arguments

port

port to wait on

timeout

time to wait (in jiffies)

Description

This function waits for up to *timeout* jiffies for an interrupt to occur on a parallel port. If the port timeout is set to zero, it returns immediately.

If an interrupt occurs before the timeout period elapses, this function returns zero immediately. If it times out, it returns one. An error code less than zero indicates an error (most likely a pending signal), and the calling code should finish what it's doing as soon as it can.

Name

parport_wait_peripheral — wait for status lines to change in 35ms

Synopsis

```
int parport_wait_peripheral (port,
                             mask,
                             result);

struct parport *  port;
unsigned char     mask;
unsigned char     result;
```

Arguments

port

port to watch

mask

status lines to watch

result

desired values of chosen status lines

Description

This function waits until the masked status lines have the desired values, or until 35ms have elapsed (see IEEE 1284-1994 page 24 to 25 for why this value in particular is hardcoded). The *mask* and *result* parameters are bitmasks, with the bits defined by the constants in parport.h: PARPORT_STATUS_BUSY, and so on.

The port is polled quickly to start off with, in anticipation of a fast response from the peripheral. This fast polling time is configurable (using /proc), and defaults to 500usec. If the timeout for this port (see parport_set_timeout) is zero, the fast polling time is 35ms, and this function does not call schedule.

If the timeout for this port is non-zero, after the fast polling fails it uses parport_wait_event to wait for up to 10ms, waking up if an interrupt occurs.

Name

parport_negotiate — negotiate an IEEE 1284 mode

Synopsis

```
int parport_negotiate (port,
                       mode);

struct parport *  port;
int               mode;
```

Arguments

port

port to use

mode

mode to negotiate to

Description

Use this to negotiate to a particular IEEE 1284 transfer mode. The *mode* parameter should be one of the constants in parport.h starting IEEE1284_MODE_xxx.

The return value is 0 if the peripheral has accepted the negotiation to the mode specified, -1 if the peripheral is not IEEE 1284 compliant (or not present), or 1 if the peripheral has rejected the negotiation.

Name

parport_write — write a block of data to a parallel port

Synopsis

```
ssize_t parport_write (port,
                       buffer,
                       len);

struct parport *  port;
const void *      buffer;
size_t            len;
```

Arguments

port

port to write to

buffer

data buffer (in kernel space)

len

number of bytes of data to transfer

Description

This will write up to *len* bytes of *buffer* to the port specified, using the IEEE 1284 transfer mode most recently negotiated to (using `parport_negotiate`), as long as that mode supports forward transfers (host to peripheral).

It is the caller's responsibility to ensure that the first *len* bytes of *buffer* are valid.

This function returns the number of bytes transferred (if zero or positive), or else an error code.

Name

parport_read — read a block of data from a parallel port

Synopsis

```
ssize_t parport_read (port,
                      buffer,
                      len);

struct parport *  port;
void *            buffer;
size_t            len;
```

Arguments

port

port to read from

buffer

data buffer (in kernel space)

len

number of bytes of data to transfer

Description

This will read up to *len* bytes of *buffer* to the port specified, using the IEEE 1284 transfer mode most recently negotiated to (using parport_negotiate), as long as that mode supports reverse transfers (peripheral to host).

It is the caller's responsibility to ensure that the first *len* bytes of *buffer* are available to write to.

This function returns the number of bytes transferred (if zero or positive), or else an error code.

Name

parport_set_timeout — set the inactivity timeout for a device

Synopsis

```
long parport_set_timeout (dev,
                          inactivity);

struct pardevice *  dev;
long                inactivity;
```

Arguments

dev

device on a port

inactivity

inactivity timeout (in jiffies)

Description

This sets the inactivity timeout for a particular device on a port. This affects functions like `parport_wait_peripheral`. The special value 0 means not to call `schedule` while dealing with this device.

The return value is the previous inactivity timeout.

Any callers of `parport_wait_event` for this device are woken up.

Name

parport_register_driver — register a parallel port device driver

Synopsis

```
int parport_register_driver (drv);

struct parport_driver *  drv;
```

Arguments

drv

structure describing the driver

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.

The *drv* structure is allocated by the caller and must not be deallocated until after calling `parport_unregister_driver`.

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. Currently it always succeeds.

Name

parport_unregister_driver — deregister a parallel port device driver

Synopsis

```
void parport_unregister_driver (drv);

struct parport_driver *  drv;
```

Arguments

drv

structure describing the driver that was given to `parport_register_driver`

Description

This should be called by a parallel port device driver that has registered itself using `parport_register_driver` when it is about to be unloaded.

When it returns, the driver's `attach` routine will no longer be called, and for each port that `attach` was called for, the `detach` routine will have been called.

All the driver's `attach` and `detach` calls are guaranteed to have finished by the time this function returns.

Name

parport_get_port — increment a port's reference count

Synopsis

```
struct parport * parport_get_port (port);

struct parport *  port;
```

Arguments

port

the port

Description

This ensures that a struct parport pointer remains valid until the matching `parport_put_port` call.

Name

parport_put_port — decrement a port's reference count

Synopsis

```
void parport_put_port (port);

struct parport *  port;
```

Arguments

port

the port

Description

This should be called once for each call to `parport_get_port`, once the port is no longer needed.

Name

parport_register_port — register a parallel port

Synopsis

```
struct parport * parport_register_port (base,
                                        irq,
                                        dma,
                                        ops);

unsigned long               base;
int                         irq;
int                         dma;
struct parport_operations *  ops;
```

Arguments

base

base I/O address

irq

IRQ line

dma

DMA channel

ops

pointer to the port driver's port operations structure

Description

When a parallel port (lowlevel) driver finds a port that should be made available to parallel port device drivers, it should call parport_register_port. The *base*, *irq*, and *dma* parameters are for the convenience of port drivers, and for ports where they aren't meaningful needn't be set to anything special. They can be altered afterwards by adjusting the relevant members of the parport structure that is returned and represents the port. They should not be tampered with after calling parport_announce_port, however.

If there are parallel port device drivers in the system that have registered themselves using parport_register_driver, they are not told about the port at this time; that is done by parport_announce_port.

The *ops* structure is allocated by the caller, and must not be deallocated before calling parport_remove_port.

If there is no memory to allocate a new parport structure, this function will return NULL.

Name

parport_announce_port — tell device drivers about a parallel port

Synopsis

```
void parport_announce_port (port);

struct parport *  port;
```

Arguments

port

parallel port to announce

Description

After a port driver has registered a parallel port with parport_register_port, and performed any necessary initialisation or adjustments, it should call parport_announce_port in order to notify all device drivers that have called parport_register_driver. Their attach functions will be called, with *port* as the parameter.

Name

parport_remove_port — deregister a parallel port

Synopsis

```
void parport_remove_port (port);

struct parport *  port;
```

Arguments

port

parallel port to deregister

Description

When a parallel port driver is forcibly unloaded, or a parallel port becomes inaccessible, the port driver must call this function in order to deal with device drivers that still want to use it.

The parport structure associated with the port has its operations structure replaced with one containing 'null' operations that return errors or just don't do anything.

Any drivers that have registered themselves using `parport_register_driver` are notified that the port is no longer accessible by having their `detach` routines called with *`port`* as the parameter.

Name

parport_register_device — register a device on a parallel port

Synopsis

```
struct pardevice * parport_register_device (port,
                                            name,
                                            pf,
                                            kf,
                                            irq_func,
                                            flags,
                                            handle);

struct parport *  port;
const char *      name;
int (*            pf(void *);
void (*           kf(void *);
void (*           irq_func(void *);
int               flags;
void *            handle;
```

Arguments

port

port to which the device is attached

name

a name to refer to the device

pf

preemption callback

kf

kick callback (wake-up)

irq_func

interrupt handler

flags

registration flags

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.

Name

parport_unregister_device — deregister a device on a parallel port

Synopsis

```
void parport_unregister_device (dev);

struct pardevice *  dev;
```

Arguments

dev

pointer to structure representing device

Description

This undoes the effect of parport_register_device.

Name

parport_find_number — find a parallel port by number

Synopsis

```
struct parport * parport_find_number (number);
```

```
int number;
```

Arguments

number

parallel port number

Description

This returns the parallel port with the specified number, or NULL if there is none.

There is an implicit parport_get_port done already; to throw away the reference to the port that parport_find_number gives you, use parport_put_port.

Name

parport_find_base — find a parallel port by base address

Synopsis

```
struct parport * parport_find_base (base);

unsigned long base;
```

Arguments

base

base I/O address

Description

This returns the parallel port with the specified base address, or NULL if there is none.

There is an implicit parport_get_port done already; to throw away the reference to the port that parport_find_base gives you, use parport_put_port.

Name

parport_claim — claim access to a parallel port device

Synopsis

```
int parport_claim (dev);

struct pardevice *  dev;
```

Arguments

dev

pointer to structure representing a device on the port

Description

This function will not block and so can be used from interrupt context. If `parport_claim` succeeds in claiming access to the port it returns zero and the port is available to use. It may fail (returning non-zero) if the port is in use by another driver and that driver is not willing to relinquish control of the port.

Name

parport_claim_or_block — claim access to a parallel port device

Synopsis

```
int parport_claim_or_block (dev);

struct pardevice *  dev;
```

Arguments

dev

pointer to structure representing a device on the port

Description

This behaves like `parport_claim`, but will block if necessary to wait for the port to be free. A return value of 1 indicates that it slept; 0 means that it succeeded without needing to sleep. A negative error code indicates failure.

Name

parport_release — give up access to a parallel port device

Synopsis

```
void parport_release (dev);

struct pardevice *  dev;
```

Arguments

dev

pointer to structure representing parallel port device

Description

This function cannot fail, but it should not be called without the port claimed. Similarly, if the port is already claimed you should not try claiming it again.

Name

parport_open — find a device by canonical device number

Synopsis

```
struct pardevice * parport_open (devnum,
                                 name);

int           devnum;
const char *  name;
```

Arguments

devnum

canonical device number

name

name to associate with the device

Description

This function is similar to `parport_register_device`, except that it locates a device by its number rather than by the port it is attached to.

All parameters except for *devnum* are the same as for `parport_register_device`. The return value is the same as for `parport_register_device`.

Name

parport_close — close a device opened with `parport_open`

Synopsis

```
void parport_close (dev);

struct pardevice *  dev;
```

Arguments

dev

device to close

Description

This is to `parport_open` as `parport_unregister_device` is to `parport_register_device`.

Chapter 4. Message-based devices

Table of Contents

Fusion message devices

Name

mpt_register — Register protocol-specific main callback handler.

Synopsis

```
u8 mpt_register (cbfunc,
                 dclass);

MPT_CALLBACK      cbfunc;
MPT_DRIVER_CLASS  dclass;
```

Arguments

cbfunc

callback function pointer

dclass

Protocol driver's class (MPT_DRIVER_CLASS enum value)

Description

This routine is called by a protocol-specific driver (SCSI host, LAN, SCSI target) to register its reply callback routine. Each protocol-specific driver must do this before it will be able to use any IOC resources, such as obtaining request frames.

NOTES

The SCSI protocol driver currently calls this routine thrice in order to register separate callbacks; one for “normal” SCSI IO; one for MptScsiTaskMgmt requests; one for Scan/DV requests.

Returns u8 valued “handle” in the range (and S.O.D. order) {N,...,7,6,5,...,1} if successful. A return value of MPT_MAX_PROTOCOL_DRIVERS (including zero!) should be considered an error by the caller.

Name

mpt_deregister — Deregister a protocol drivers resources.

Synopsis

```
void mpt_deregister (cb_idx);

u8  cb_idx;
```

Arguments

cb_idx

previously registered callback handle

Description

Each protocol-specific driver should call this routine when its module is unloaded.

Name

mpt_event_register — Register protocol-specific event callback handler.

Synopsis

```
int mpt_event_register (cb_idx,
                        ev_cbfunc);

u8             cb_idx;
MPT_EVHANDLER  ev_cbfunc;
```

Arguments

cb_idx

previously registered (via mpt_register) callback handle

ev_cbfunc

callback function

Description

This routine can be called by one or more protocol-specific drivers if/when they choose to be notified of MPT events.

Returns 0 for success.

Name

mpt_event_deregister — Deregister protocol-specific event callback handler

Synopsis

```
void mpt_event_deregister (cb_idx);

u8  cb_idx;
```

Arguments

cb_idx

previously registered callback handle

Description

Each protocol-specific driver should call this routine when it does not (or can no longer) handle events, or when its module is unloaded.

Name

mpt_reset_register — Register protocol-specific IOC reset handler.

Synopsis

```
int mpt_reset_register (cb_idx,
                        reset_func);

u8                cb_idx;
MPT_RESETHANDLER  reset_func;
```

Arguments

cb_idx

previously registered (via mpt_register) callback handle

reset_func

reset function

Description

This routine can be called by one or more protocol-specific drivers if/when they choose to be notified of IOC resets.

Returns 0 for success.

Name

mpt_reset_deregister — Deregister protocol-specific IOC reset handler.

Synopsis

```
void mpt_reset_deregister (cb_idx);

u8  cb_idx;
```

Arguments

cb_idx

previously registered callback handle

Description

Each protocol-specific driver should call this routine when it does not (or can no longer) handle IOC reset handling, or when its module is unloaded.

Name

mpt_device_driver_register — Register device driver hooks

Synopsis

```
int mpt_device_driver_register (dd_cbfunc,
                                cb_idx);

struct mpt_pci_driver *  dd_cbfunc;
u8                       cb_idx;
```

Arguments

dd_cbfunc

driver callbacks struct

cb_idx

MPT protocol driver index

Name

mpt_device_driver_deregister — DeRegister device driver hooks

Synopsis

```
void mpt_device_driver_deregister (cb_idx);

u8  cb_idx;
```

Arguments

cb_idx

MPT protocol driver index

Name

mpt_get_msg_frame — Obtain an MPT request frame from the pool

Synopsis

```
MPT_FRAME_HDR* mpt_get_msg_frame (cb_idx,
                                  ioc);

u8             cb_idx;
MPT_ADAPTER *  ioc;
```

Arguments

cb_idx

Handle of registered MPT protocol driver

ioc

Pointer to MPT adapter structure

Description

Obtain an MPT request frame from the pool (of 1024) that are allocated per MPT adapter.

Returns pointer to a MPT request frame or NULL if none are available or IOC is not active.

Name

mpt_put_msg_frame — Send a protocol-specific MPT request frame to an IOC

Synopsis

```
void mpt_put_msg_frame (cb_idx,
                        ioc,
                        mf);

u8               cb_idx;
MPT_ADAPTER *    ioc;
MPT_FRAME_HDR *  mf;
```

Arguments

cb_idx

Handle of registered MPT protocol driver

ioc

Pointer to MPT adapter structure

mf

Pointer to MPT request frame

Description

This routine posts an MPT request frame to the request post FIFO of a specific MPT adapter.

Name

mpt_put_msg_frame_hi_pri — Send a hi-pri protocol-specific MPT request frame

Synopsis

```
void mpt_put_msg_frame_hi_pri (cb_idx,
                               ioc,
                               mf);

u8              cb_idx;
MPT_ADAPTER *   ioc;
MPT_FRAME_HDR * mf;
```

Arguments

cb_idx

Handle of registered MPT protocol driver

ioc

Pointer to MPT adapter structure

mf

Pointer to MPT request frame

Description

Send a protocol-specific MPT request frame to an IOC using hi-priority request queue.

This routine posts an MPT request frame to the request post FIFO of a specific MPT adapter.

Name

mpt_free_msg_frame — Place MPT request frame back on FreeQ.

Synopsis

```
void mpt_free_msg_frame (ioc,
                         mf);

MPT_ADAPTER *    ioc;
MPT_FRAME_HDR *  mf;
```

Arguments

ioc

Pointer to MPT adapter structure

mf

Pointer to MPT request frame

Description

This routine places a MPT request frame back on the MPT adapter's FreeQ.

Name

mpt_send_handshake_request — Send MPT request via doorbell handshake method.

Synopsis

```
int mpt_send_handshake_request (cb_idx,
                                ioc,
                                reqBytes,
                                req,
                                sleepFlag);

u8             cb_idx;
MPT_ADAPTER *  ioc;
int            reqBytes;
u32 *          req;
int            sleepFlag;
```

Arguments

cb_idx

Handle of registered MPT protocol driver

ioc

Pointer to MPT adapter structure

reqBytes

Size of the request in bytes

req

Pointer to MPT request frame

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.

Name

mpt_verify_adapter — Given IOC identifier, set pointer to its adapter structure.

Synopsis

```
int mpt_verify_adapter (iocid,
                        iocpp);

int             iocid;
MPT_ADAPTER **  iocpp;
```

Arguments

iocid

IOC unique identifier (integer)

iocpp

Pointer to pointer to IOC adapter

Description

Given a unique IOC identifier, set pointer to the associated MPT adapter structure.

Returns iocid and sets iocpp if iocid is found. Returns -1 if iocid is not found.

Name

mpt_attach — Install a PCI intelligent MPT adapter.

Synopsis

```
int mpt_attach (pdev,
                id);

struct pci_dev *              pdev;
const struct pci_device_id *  id;
```

Arguments

pdev

Pointer to pci_dev structure

id

PCI device ID information

Description

This routine performs all the steps necessary to bring the IOC of a MPT adapter to a OPERATIONAL state. This includes registering memory regions, registering the interrupt, and allocating request and reply memory pools.

This routine also pre-fetches the LAN MAC address of a Fibre Channel MPT adapter.

Returns 0 for success, non-zero for failure.

TODO

Add support for polled controllers

Name

mpt_detach — Remove a PCI intelligent MPT adapter.

Synopsis

```
void mpt_detach (pdev);

struct pci_dev *  pdev;
```

Arguments

pdev

Pointer to pci_dev structure

Name

mpt_suspend — Fusion MPT base driver suspend routine.

Synopsis

```
int mpt_suspend (pdev,
                 state);

struct pci_dev *  pdev;
pm_message_t      state;
```

Arguments

pdev

Pointer to pci_dev structure

state

new state to enter

Name

mpt_resume — Fusion MPT base driver resume routine.

Synopsis

```
int mpt_resume (pdev);

struct pci_dev *  pdev;
```

Arguments

pdev

Pointer to pci_dev structure

Name

mpt_GetIocState — Get the current state of a MPT adapter.

Synopsis

```
u32 mpt_GetIocState (ioc,
                     cooked);

MPT_ADAPTER *  ioc;
int            cooked;
```

Arguments

ioc

Pointer to MPT_ADAPTER structure

cooked

Request raw or cooked IOC state

Description

Returns all IOC Doorbell register bits if cooked==0, else just the Doorbell bits in MPI_IOC_STATE_MASK.

Name

mpt_alloc_fw_memory — allocate firmware memory

Synopsis

```
int mpt_alloc_fw_memory (ioc,
                         size);

MPT_ADAPTER *  ioc;
int            size;
```

Arguments

ioc

Pointer to MPT_ADAPTER structure

size

total FW bytes

Description

If memory has already been allocated, the same (cached) value is returned.

Return 0 if successfull, or non-zero for failure

Name

mpt_free_fw_memory — free firmware memory

Synopsis

```
void mpt_free_fw_memory (ioc);

MPT_ADAPTER *  ioc;
```

Arguments

ioc

Pointer to MPT_ADAPTER structure

Description

If alt_img is NULL, delete from ioc structure. Else, delete a secondary image in same format.