

# **Voltage and current regulator API**

**Liam Girdwood**

**`lrg@slimlogic.co.uk`**

**Mark Brown**

**Wolfson Microelectronics**

**`broonie@opensource.wolfsonmicro.com`**

# **Voltage and current regulator API**

by Liam Girdwood and Mark Brown

Copyright © 2007-2008 Wolfson Microelectronics

Copyright © 2008 Liam Girdwood

This documentation is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License version 2 as published by the Free Software Foundation.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

For more details see the file COPYING in the source distribution of Linux.

# Table of Contents

<b>1. Introduction.....</b>	<b>1</b>
1.1. Glossary .....	1
Glossary .....	1
<b>2. Consumer driver interface .....</b>	<b>3</b>
2.1. Enabling and disabling.....	3
2.2. Configuration .....	3
2.3. Callbacks.....	3
<b>3. Regulator driver interface.....</b>	<b>5</b>
<b>4. Machine interface.....</b>	<b>7</b>
4.1. Supplies.....	7
4.2. Constraints .....	7
<b>5. API reference.....</b>	<b>9</b>
struct regulator_bulk_data.....	9
struct regulator_state .....	10
struct regulation_constraints .....	11
struct regulator_consumer_supply .....	13
struct regulator_init_data .....	14
struct regulator_ops.....	15
struct regulator_desc .....	18
regulator_get .....	19
devm_regulator_get .....	20
regulator_get_exclusive .....	21
regulator_put.....	22
devm_regulator_put .....	23
regulator_enable.....	24
regulator_disable.....	25
regulator_force_disable.....	26
regulator_disable_deferred .....	27
regulator_is_enabled .....	28
regulator_count_voltages .....	29
regulator_list_voltage.....	30
regulator_is_supported_voltage .....	31
regulator_set_voltage .....	32
regulator_set_voltage_time.....	33
regulator_sync_voltage .....	34
regulator_get_voltage.....	35
regulator_set_current_limit.....	36
regulator_get_current_limit .....	37
regulator_set_mode.....	38
regulator_get_mode .....	39

regulator_set_optimum_mode .....	40
regulator_register_notifier.....	41
regulator_unregister_notifier.....	42
regulator_bulk_get .....	43
devm_regulator_bulk_get .....	44
regulator_bulk_enable.....	45
regulator_bulk_disable.....	46
regulator_bulk_force_disable.....	47
regulator_bulk_free.....	48
regulator_notifier_call_chain .....	49
regulator_mode_to_status .....	50
regulator_register .....	51
regulator_unregister .....	52
regulator_suspend_prepare .....	53
regulator_suspend_finish .....	54
regulator_has_full_constraints .....	55
regulator_use_dummy_regulator .....	56
rdev_get_drvdata.....	57
regulator_get_drvdata .....	57
regulator_set_drvdata.....	58
rdev_get_id .....	59

# Chapter 1. Introduction

This framework is designed to provide a standard kernel interface to control voltage and current regulators.

The intention is to allow systems to dynamically control regulator power output in order to save power and prolong battery life. This applies to both voltage regulators (where voltage output is controllable) and current sinks (where current limit is controllable).

Note that additional (and currently more complete) documentation is available in the Linux kernel source under `Documentation/power/regulator`.

## 1.1. Glossary

The regulator API uses a number of terms which may not be familiar:

## Glossary

### Regulator

Electronic device that supplies power to other devices. Most regulators can enable and disable their output and some can also control their output voltage or current.

### Consumer

Electronic device which consumes power provided by a regulator. These may either be static, requiring only a fixed supply, or dynamic, requiring active management of the regulator at runtime.

### Power Domain

The electronic circuit supplied by a given regulator, including the regulator and all consumer devices. The configuration of the regulator is shared between all the components in the circuit.

## **Power Management Integrated Circuit**

An IC which contains numerous regulators and often also other subsystems. In an embedded system the primary PMIC is often equivalent to a combination of the PSU and southbridge in a desktop system.

# Chapter 2. Consumer driver interface

This offers a similar API to the kernel clock framework. Consumer drivers use get and put operations to acquire and release regulators. Functions are provided to enable and disable the regulator and to get and set the runtime parameters of the regulator.

When requesting regulators consumers use symbolic names for their supplies, such as "Vcc", which are mapped into actual regulator devices by the machine interface.

A stub version of this API is provided when the regulator framework is not in use in order to minimise the need to use `ifdefs`.

## 2.1. Enabling and disabling

The regulator API provides reference counted enabling and disabling of regulators. Consumer devices use the `regulator_enable` and `regulator_disable` functions to enable and disable regulators. Calls to the two functions must be balanced.

Note that since multiple consumers may be using a regulator and machine constraints may not allow the regulator to be disabled there is no guarantee that calling `regulator_disable` will actually cause the supply provided by the regulator to be disabled. Consumer drivers should assume that the regulator may be enabled at all times.

## 2.2. Configuration

Some consumer devices may need to be able to dynamically configure their supplies. For example, MMC drivers may need to select the correct operating voltage for their cards. This may be done while the regulator is enabled or disabled.

The `regulator_set_voltage` and `regulator_set_current_limit` functions provide the primary interface for this. Both take ranges of voltages and currents, supporting drivers that do not require a specific value (eg, CPU frequency scaling normally permits the CPU to use a wider range of supply voltages at lower frequencies but does not require that the supply voltage be lowered). Where an exact value is required both minimum and maximum values should be identical.

## **2.3. Callbacks**

Callbacks may also be registered for events such as regulation failures.



# Chapter 3. Regulator driver interface

Drivers for regulator chips register the regulators with the regulator core, providing operations structures to the core. A notifier interface allows error conditions to be reported to the core.

Registration should be triggered by explicit setup done by the platform, supplying a struct `regulator_init_data` for the regulator containing constraint and supply information.



# Chapter 4. Machine interface

This interface provides a way to define how regulators are connected to consumers on a given system and what the valid operating parameters are for the system.

## 4.1. Supplies

Regulator supplies are specified using struct `regulator_consumer_supply`. This is done at driver registration time as part of the machine constraints.

## 4.2. Constraints

As well as defining the connections the machine interface also provides constraints defining the operations that clients are allowed to perform and the parameters that may be set. This is required since generally regulator devices will offer more flexibility than it is safe to use on a given system, for example supporting higher supply voltages than the consumers are rated for.

This is done at driver registration time by providing a struct `regulation_constraints`.

The constraints may also specify an initial configuration for the regulator in the constraints, which is particularly useful for use with static consumers.



# Chapter 5. API reference

Due to limitations of the kernel documentation framework and the existing layout of the source code the entire regulator API is documented here.

## struct regulator\_bulk\_data

### LINUX

Kernel Hackers Manual January 2013

### Name

`struct regulator_bulk_data` — Data used for bulk regulator operations.

### Synopsis

```
struct regulator_bulk_data {
    const char * supply;
    struct regulator * consumer;
};
```

### Members

`supply`

The name of the supply. Initialised by the user before using the bulk regulator APIs.

`consumer`

The regulator consumer for the supply. This will be managed by the bulk API.

### Description

The regulator APIs provide a series of `regulator_bulk_` API calls as a convenience to consumers which require multiple supplies. This structure is used to manage data for these calls.

# struct regulator\_state

## LINUX

Kernel Hackers Manual January 2013

### Name

`struct regulator_state` — regulator state during low power system states

### Synopsis

```
struct regulator_state {  
    int uV;  
    unsigned int mode;  
    int enabled;  
    int disabled;  
};
```

### Members

`uV`

Operating voltage during suspend.

`mode`

Operating mode during suspend.

`enabled`

Enabled during suspend.

`disabled`

Disabled during suspend.

## Description

This describes a regulators state during a system wide low power state. One of enabled or disabled must be set for the configuration to be applied.

## struct regulation\_constraints

### LINUX

Kernel Hackers Manual January 2013

### Name

`struct regulation_constraints` — regulator operating constraints.

### Synopsis

```
struct regulation_constraints {
    const char * name;
    int min_uV;
    int max_uV;
    int uV_offset;
    int min_uA;
    int max_uA;
    unsigned int valid_modes_mask;
    unsigned int valid_ops_mask;
    int input_uV;
    struct regulator_state state_disk;
    struct regulator_state state_mem;
    struct regulator_state state_standby;
    suspend_state_t initial_state;
    unsigned int initial_mode;
    unsigned always_on:1;
    unsigned boot_on:1;
    unsigned apply_uV:1;
};
```

## Members

name

Descriptive name for the constraints, used for display purposes.

min\_uV

Smallest voltage consumers may set.

max\_uV

Largest voltage consumers may set.

uV\_offset

Offset applied to voltages from consumer to compensate for voltage drops.

min\_uA

Smallest current consumers may set.

max\_uA

Largest current consumers may set.

valid\_modes\_mask

Mask of modes which may be configured by consumers.

valid\_ops\_mask

Operations which may be performed by consumers.

input\_uV

Input voltage for regulator when supplied by another regulator.

state\_disk

State for regulator when system is suspended in disk mode.

state\_mem

State for regulator when system is suspended in mem mode.

state\_standby

State for regulator when system is suspended in standby mode.

initial\_state

Suspend state to set by default.



`initial_mode`

Mode to set at startup.

`always_on`

Set if the regulator should never be disabled.

`boot_on`

Set if the regulator is enabled when the system is initially started. If the regulator is not enabled by the hardware or bootloader then it will be enabled when the constraints are applied.

`apply_uV`

Apply the voltage constraint when initialising.

## Description

This struct describes regulator and board/machine specific constraints.

# struct regulator\_consumer\_supply

## LINUX

Kernel Hackers Manual January 2013

## Name

`struct regulator_consumer_supply` — supply -> device mapping

## Synopsis

```
struct regulator_consumer_supply {  
    const char * dev_name;  
    const char * supply;  
};
```

## Members

`dev_name`

Result of `dev_name` for the consumer.

`supply`

Name for the supply.

## Description

This maps a supply name to a device. Use of `dev_name` allows support for buses which make struct device available late such as I2C.

# struct regulator\_init\_data

## LINUX

Kernel Hackers Manual January 2013

## Name

`struct regulator_init_data` — regulator platform initialisation data.

## Synopsis

```
struct regulator_init_data {  
    const char * supply_regulator;  
    struct regulation_constraints constraints;  
    int num_consumer_supplies;  
    struct regulator_consumer_supply * consumer_supplies;  
    int (* regulator_init) (void *driver_data);  
    void * driver_data;  
};
```

## Members

`supply_regulator`

Parent regulator. Specified using the regulator name as it appears in the name field in sysfs, which can be explicitly set using the constraints field 'name'.

`constraints`

Constraints. These must be specified for the regulator to be usable.

`num_consumer_supplies`

Number of consumer device supplies.

`consumer_supplies`

Consumer device supply configuration.

`regulator_init`

Callback invoked when the regulator has been registered.

`driver_data`

Data passed to `regulator_init`.

## Description

Initialisation constraints, our supply and consumers supplies.

## struct regulator\_ops

**LINUX**

Kernel Hackers Manual January 2013

## Name

`struct regulator_ops` — regulator operations.

## Synopsis

```
struct regulator_ops {
    int (* list_voltage) (struct regulator_dev *, unsigned selector);
    int (* set_voltage) (struct regulator_dev *, int min_uV, int max_uV, unsigned selector);
    int (* set_voltage_sel) (struct regulator_dev *, unsigned selector);
    int (* get_voltage) (struct regulator_dev *);
    int (* get_voltage_sel) (struct regulator_dev *);
    int (* set_current_limit) (struct regulator_dev *, int min_uA, int max_uA, unsigned selector);
    int (* get_current_limit) (struct regulator_dev *);
    int (* enable) (struct regulator_dev *);
    int (* disable) (struct regulator_dev *);
    int (* is_enabled) (struct regulator_dev *);
    int (* set_mode) (struct regulator_dev *, unsigned int mode);
    unsigned int (* get_mode) (struct regulator_dev *);
    int (* enable_time) (struct regulator_dev *);
    int (* set_voltage_time_sel) (struct regulator_dev *, unsigned int old_selector, unsigned int new_selector);
    int (* get_status) (struct regulator_dev *);
    unsigned int (* get_optimum_mode) (struct regulator_dev *, int input_uV);
    int (* set_suspend_voltage) (struct regulator_dev *, int uV);
    int (* set_suspend_enable) (struct regulator_dev *);
    int (* set_suspend_disable) (struct regulator_dev *);
    int (* set_suspend_mode) (struct regulator_dev *, unsigned int mode);
};
```

## Members

### list\_voltage

Return one of the supported voltages, in microvolts; zero if the selector indicates a voltage that is unusable on this system; or negative errno. Selectors range from zero to one less than regulator\_desc.n\_voltages. Voltages may be reported in any order.

### set\_voltage

Set the voltage for the regulator within the range specified. The driver should select the voltage closest to min\_uV.

### set\_voltage\_sel

Set the voltage for the regulator using the specified selector.

### get\_voltage

Return the currently configured voltage for the regulator.

`get_voltage_sel`

Return the currently configured voltage selector for the regulator.

`set_current_limit`

Configure a limit for a current-limited regulator.

`get_current_limit`

Get the configured limit for a current-limited regulator.

`enable`

Configure the regulator as enabled.

`disable`

Configure the regulator as disabled.

`is_enabled`

Return 1 if the regulator is enabled, 0 if not. May also return negative `errno`.

`set_mode`

Set the configured operating mode for the regulator.

`get_mode`

Get the configured operating mode for the regulator.

`enable_time`

Time taken for the regulator voltage output voltage to stabilise after being enabled, in microseconds.

`set_voltage_time_sel`

Time taken for the regulator voltage output voltage to stabilise after being set to a new value, in microseconds. The function provides the from and to voltage selector, the function should return the worst case.

`get_status`

Return actual (not as-configured) status of regulator, as a `REGULATOR_STATUS` value (or negative `errno`)

`get_optimum_mode`

Get the most efficient operating mode for the regulator when running with the specified parameters.

`set_suspend_voltage`

Set the voltage for the regulator when the system is suspended.

`set_suspend_enable`

Mark the regulator as enabled when the system is suspended.

`set_suspend_disable`

Mark the regulator as disabled when the system is suspended.

`set_suspend_mode`

Set the operating mode for the regulator when the system is suspended.

## Description

This struct describes regulator operations which can be implemented by regulator chip drivers.

# struct regulator\_desc

## LINUX

Kernel Hackers Manual January 2013

## Name

`struct regulator_desc` — Regulator descriptor

## Synopsis

```
struct regulator_desc {
    const char * name;
    const char * supply_name;
    int id;
    unsigned n_voltages;
    struct regulator_ops * ops;
    int irq;
    enum regulator_type type;
```

```
struct module * owner;  
};
```

## Members

`name`

Identifying name for the regulator.

`supply_name`

Identifying the regulator supply

`id`

Numerical identifier for the regulator.

`n_voltages`

Number of selectors available for `ops.list_voltage`.

`ops`

Regulator operations table.

`irq`

Interrupt number for the regulator.

`type`

Indicates if the regulator is a voltage or current regulator.

`owner`

Module providing the regulator, used for refcounting.

## Description

Each regulator registered with the core is described with a structure of this type.

# regulator\_get

## LINUX

Kernel Hackers Manual January 2013

### Name

`regulator_get` — lookup and obtain a reference to a regulator.

### Synopsis

```
struct regulator * regulator_get (struct device * dev, const  
char * id);
```

### Arguments

*dev*

device for regulator “consumer”

*id*

Supply name or regulator ID.

### Description

Returns a struct regulator corresponding to the regulator producer, or `IS_ERR` condition containing errno.

Use of supply names configured via `regulator_set_device_supply` is strongly encouraged. It is recommended that the supply name used should match the name used for the supply and/or the relevant device pins in the datasheet.



# devm\_regulator\_get

## LINUX

Kernel Hackers Manual January 2013

### Name

devm\_regulator\_get — Resource managed regulator\_get

### Synopsis

```
struct regulator * devm_regulator_get (struct device * dev,
const char * id);
```

### Arguments

*dev*

device for regulator “consumer”

*id*

Supply name or regulator ID.

### Description

Managed regulator\_get. Regulators returned from this function are automatically regulator\_put on driver detach. See regulator\_get for more information.

# regulator\_get\_exclusive

## LINUX

## Name

`regulator_get_exclusive` — obtain exclusive access to a regulator.

## Synopsis

```
struct regulator * regulator_get_exclusive (struct device *  
dev, const char * id);
```

## Arguments

*dev*

device for regulator “consumer”

*id*

Supply name or regulator ID.

## Description

Returns a struct regulator corresponding to the regulator producer, or `IS_ERR` condition containing `errno`. Other consumers will be unable to obtain this reference is held and the use count for the regulator will be initialised to reflect the current state of the regulator.

This is intended for use by consumers which cannot tolerate shared use of the regulator such as those which need to force the regulator off for correct operation of the hardware they are controlling.

Use of supply names configured via `regulator_set_device_supply` is strongly encouraged. It is recommended that the supply name used should match the name used for the supply and/or the relevant device pins in the datasheet.

# regulator\_put

## LINUX

Kernel Hackers Manual January 2013

### Name

`regulator_put` — "free" the regulator source

### Synopsis

```
void regulator_put (struct regulator * regulator);
```

### Arguments

*regulator*

regulator source

### Note

drivers must ensure that all `regulator_enable` calls made on this regulator source are balanced by `regulator_disable` calls prior to calling this function.

# devm\_regulator\_put

## LINUX

## Name

`devm_regulator_put` — Resource managed `regulator_put`

## Synopsis

```
void devm_regulator_put (struct regulator * regulator);
```

## Arguments

*regulator*

regulator to free

## Description

Deallocate a regulator allocated with `devm_regulator_get`. Normally this function will not need to be called and the resource management code will ensure that the resource is freed.

# regulator\_enable

## LINUX

## Name

`regulator_enable` — enable regulator output

## Synopsis

```
int regulator_enable (struct regulator * regulator);
```

## Arguments

*regulator*

regulator source

## Description

Request that the regulator be enabled with the regulator output at the predefined voltage or current value. Calls to `regulator_enable` must be balanced with calls to `regulator_disable`.

## NOTE

the output value can be set by other drivers, boot loader or may be hardwired in the regulator.

# regulator\_disable

## LINUX

Kernel Hackers Manual January 2013

## Name

`regulator_disable` — disable regulator output

## Synopsis

```
int regulator_disable (struct regulator * regulator);
```

## Arguments

*regulator*

regulator source

## Description

Disable the regulator output voltage or current. Calls to `regulator_enable` must be balanced with calls to `regulator_disable`.

## NOTE

this will only disable the regulator output if no other consumer devices have it enabled, the regulator device supports disabling and machine constraints permit this operation.

# regulator\_force\_disable

## LINUX

Kernel Hackers Manual January 2013

## Name

`regulator_force_disable` — force disable regulator output

## Synopsis

```
int regulator_force_disable (struct regulator * regulator);
```

## Arguments

*regulator*

regulator source

## Description

Forcibly disable the regulator output voltage or current.

## NOTE

this *\*will\** disable the regulator output even if other consumer devices have it enabled. This should be used for situations when device damage will likely occur if the regulator is not disabled (e.g. over temp).

# regulator\_disable\_deferred

## LINUX

Kernel Hackers Manual January 2013

## Name

`regulator_disable_deferred` — disable regulator output with delay

## Synopsis

```
int regulator_disable_deferred (struct regulator * regulator,  
int ms);
```

## Arguments

*regulator*

regulator source

*ms*

milliseconds until the regulator is disabled

## Description

Execute `regulator_disable` on the regulator after a delay. This is intended for use with devices that require some time to quiesce.

## NOTE

this will only disable the regulator output if no other consumer devices have it enabled, the regulator device supports disabling and machine constraints permit this operation.

## regulator\_is\_enabled

### LINUX

Kernel Hackers Manual January 2013

## Name

`regulator_is_enabled` — is the regulator output enabled



## Synopsis

```
int regulator_is_enabled (struct regulator * regulator);
```

## Arguments

*regulator*

regulator source

## Description

Returns positive if the regulator driver backing the source/client has requested that the device be enabled, zero if it hasn't, else a negative errno code.

Note that the device backing this regulator handle can have multiple users, so it might be enabled even if `regulator_enable` was never called for this particular source.

# regulator\_count\_voltages

## LINUX

Kernel Hackers Manual January 2013

## Name

`regulator_count_voltages` — **count** `regulator_list_voltage` selectors

## Synopsis

```
int regulator_count_voltages (struct regulator * regulator);
```

## Arguments

*regulator*

regulator source

## Description

Returns number of selectors, or negative errno. Selectors are numbered starting at zero, and typically correspond to bitfields in hardware registers.

# regulator\_list\_voltage

## LINUX

Kernel Hackers Manual January 2013

## Name

`regulator_list_voltage` — enumerate supported voltages

## Synopsis

```
int regulator_list_voltage (struct regulator * regulator,  
unsigned selector);
```

## Arguments

*regulator*

regulator source

*selector*

identify voltage to list

## Context

can sleep

## Description

Returns a voltage that can be passed to *regulator\_set\_voltage()*, zero if this selector code can't be used on this system, or a negative errno.

# regulator\_is\_supported\_voltage

## LINUX

Kernel Hackers Manual January 2013

## Name

*regulator\_is\_supported\_voltage* — check if a voltage range can be supported

## Synopsis

```
int regulator_is_supported_voltage (struct regulator *
regulator, int min_uV, int max_uV);
```

## Arguments

*regulator*

Regulator to check.

*min\_uV*

Minimum required voltage in uV.

*max\_uV*

Maximum required voltage in uV.

## Description

Returns a boolean or a negative error code.

# regulator\_set\_voltage

## LINUX

Kernel Hackers Manual January 2013

## Name

`regulator_set_voltage` — set regulator output voltage

## Synopsis

```
int regulator_set_voltage (struct regulator * regulator, int  
min_uV, int max_uV);
```

## Arguments

*regulator*

regulator source

*min\_uV*

Minimum required voltage in uV

*max\_uV*

Maximum acceptable voltage in uV

## Description

Sets a voltage regulator to the desired output voltage. This can be set during any regulator state. IOW, regulator can be disabled or enabled.

If the regulator is enabled then the voltage will change to the new value immediately otherwise if the regulator is disabled the regulator will output at the new voltage when enabled.

## NOTE

If the regulator is shared between several devices then the lowest request voltage that meets the system constraints will be used. Regulator system constraints must be set for this regulator before calling this function otherwise this call will fail.

# regulator\_set\_voltage\_time

## LINUX

Kernel Hackers Manual January 2013

## Name

`regulator_set_voltage_time` — get raise/fall time

## Synopsis

```
int regulator_set_voltage_time (struct regulator * regulator,
int old_uV, int new_uV);
```

## Arguments

*regulator*

regulator source

*old\_uV*

starting voltage in microvolts

*new\_uV*

target voltage in microvolts

## Description

Provided with the starting and ending voltage, this function attempts to calculate the time in microseconds required to rise or fall to this new voltage.

# regulator\_sync\_voltage

## LINUX

Kernel Hackers Manual January 2013

## Name

`regulator_sync_voltage` — re-apply last regulator output voltage

## Synopsis

```
int regulator_sync_voltage (struct regulator * regulator);
```

## Arguments

*regulator*

regulator source

## Description

Re-apply the last configured voltage. This is intended to be used where some external control source the consumer is cooperating with has caused the configured voltage to change.

# regulator\_get\_voltage

## LINUX

Kernel Hackers Manual January 2013

## Name

`regulator_get_voltage` — get regulator output voltage

## Synopsis

```
int regulator_get_voltage (struct regulator * regulator);
```

## Arguments

*regulator*

regulator source

## Description

This returns the current regulator voltage in uV.

## NOTE

If the regulator is disabled it will return the voltage value. This function should not be used to determine regulator state.

# regulator\_set\_current\_limit

## LINUX

Kernel Hackers Manual January 2013

## Name

`regulator_set_current_limit` — set regulator output current limit

## Synopsis

```
int regulator_set_current_limit (struct regulator * regulator,  
int min_uA, int max_uA);
```



## Arguments

*regulator*

regulator source

*min\_uA*

Minimum supported current in uA

*max\_uA*

Maximum supported current in uA

## Description

Sets current sink to the desired output current. This can be set during any regulator state. IOW, regulator can be disabled or enabled.

If the regulator is enabled then the current will change to the new value immediately otherwise if the regulator is disabled the regulator will output at the new current when enabled.

## NOTE

Regulator system constraints must be set for this regulator before calling this function otherwise this call will fail.

# regulator\_get\_current\_limit

## LINUX

Kernel Hackers Manual January 2013

## Name

`regulator_get_current_limit` — get regulator output current

## Synopsis

```
int regulator_get_current_limit (struct regulator *  
regulator);
```

## Arguments

*regulator*

regulator source

## Description

This returns the current supplied by the specified current sink in uA.

## NOTE

If the regulator is disabled it will return the current value. This function should not be used to determine regulator state.

# regulator\_set\_mode

## LINUX

Kernel Hackers Manual January 2013

## Name

`regulator_set_mode` — set regulator operating mode

## Synopsis

```
int regulator_set_mode (struct regulator * regulator, unsigned
int mode);
```

## Arguments

*regulator*

regulator source

*mode*

operating mode - one of the REGULATOR\_MODE constants

## Description

Set regulator operating mode to increase regulator efficiency or improve regulation performance.

## NOTE

Regulator system constraints must be set for this regulator before calling this function otherwise this call will fail.

# regulator\_get\_mode

## LINUX

Kernel Hackers Manual January 2013

## Name

`regulator_get_mode` — get regulator operating mode

## Synopsis

```
unsigned int regulator_get_mode (struct regulator *  
regulator);
```

## Arguments

*regulator*

regulator source

## Description

Get the current regulator operating mode.

# regulator\_set\_optimum\_mode

## LINUX

Kernel Hackers Manual January 2013

## Name

`regulator_set_optimum_mode` — set regulator optimum operating mode

## Synopsis

```
int regulator_set_optimum_mode (struct regulator * regulator,  
int uA_load);
```

## Arguments

*regulator*

regulator source

*uA\_load*

load current

## Description

Notifies the regulator core of a new device load. This is then used by DRMS (if enabled by constraints) to set the most efficient regulator operating mode for the new regulator loading.

Consumer devices notify their supply regulator of the maximum power they will require (can be taken from device datasheet in the power consumption tables) when they change operational status and hence power state. Examples of operational state changes that can affect power

## consumption are

-

- o Device is opened / closed.
- o Device I/O is about to begin or has just finished.
- o Device is idling in between work.

This information is also exported via sysfs to userspace.

DRMS will sum the total requested load on the regulator and change to the most efficient operating mode if platform constraints allow.

Returns the new regulator mode or error.

## regulator\_register\_notifier

**LINUX**

## Name

`regulator_register_notifier` — register regulator event notifier

## Synopsis

```
int regulator_register_notifier (struct regulator * regulator,
struct notifier_block * nb);
```

## Arguments

*regulator*

regulator source

*nb*

notifier block

## Description

Register notifier block to receive regulator events.

# regulator\_unregister\_notifier

## LINUX

## Name

`regulator_unregister_notifier` — unregister regulator event notifier

## Synopsis

```
int regulator_unregister_notifier (struct regulator *
regulator, struct notifier_block * nb);
```

## Arguments

*regulator*

regulator source

*nb*

notifier block

## Description

Unregister regulator event notifier block.

# regulator\_bulk\_get

## LINUX

Kernel Hackers Manual January 2013

## Name

`regulator_bulk_get` — get multiple regulator consumers

## Synopsis

```
int regulator_bulk_get (struct device * dev, int
num_consumers, struct regulator_bulk_data * consumers);
```

## Arguments

*dev*

Device to supply

*num\_consumers*

Number of consumers to register

*consumers*

Configuration of consumers; clients are stored here.

## Description

*return* 0 on success, an errno on failure.

This helper function allows drivers to get several regulator consumers in one operation. If any of the regulators cannot be acquired then any regulators that were allocated will be freed before returning to the caller.

# devm\_regulator\_bulk\_get

## LINUX

Kernel Hackers Manual January 2013

## Name

`devm_regulator_bulk_get` — managed get multiple regulator consumers

## Synopsis

```
int devm_regulator_bulk_get (struct device * dev, int
num_consumers, struct regulator_bulk_data * consumers);
```



## Arguments

*dev*

Device to supply

*num\_consumers*

Number of consumers to register

*consumers*

Configuration of consumers; clients are stored here.

## Description

*return* 0 on success, an errno on failure.

This helper function allows drivers to get several regulator consumers in one operation with management, the regulators will automatically be freed when the device is unbound. If any of the regulators cannot be acquired then any regulators that were allocated will be freed before returning to the caller.

# regulator\_bulk\_enable

## LINUX

Kernel Hackers Manual January 2013

## Name

`regulator_bulk_enable` — enable multiple regulator consumers

## Synopsis

```
int regulator_bulk_enable (int num_consumers, struct  
regulator_bulk_data * consumers);
```

## Arguments

*num\_consumers*

Number of consumers

*consumers*

Consumer data; clients are stored here. *return* 0 on success, an errno on failure

## Description

This convenience API allows consumers to enable multiple regulator clients in a single API call. If any consumers cannot be enabled then any others that were enabled will be disabled again prior to return.

# regulator\_bulk\_disable

**LINUX**

Kernel Hackers Manual January 2013

## Name

`regulator_bulk_disable` — disable multiple regulator consumers

## Synopsis

```
int regulator_bulk_disable (int num_consumers, struct
regulator_bulk_data * consumers);
```

## Arguments

*num\_consumers*

Number of consumers

*consumers*

Consumer data; clients are stored here. *return* 0 on success, an errno on failure

## Description

This convenience API allows consumers to disable multiple regulator clients in a single API call. If any consumers cannot be disabled then any others that were disabled will be enabled again prior to return.

# regulator\_bulk\_force\_disable

**LINUX**

Kernel Hackers Manual January 2013

## Name

`regulator_bulk_force_disable` — force disable multiple regulator consumers

## Synopsis

```
int regulator_bulk_force_disable (int num_consumers, struct  
regulator_bulk_data * consumers);
```

## Arguments

*num\_consumers*

Number of consumers

*consumers*

Consumer data; clients are stored here. *return* 0 on success, an errno on failure

## Description

This convenience API allows consumers to forcibly disable multiple regulator clients in a single API call.

## NOTE

This should be used for situations when device damage will likely occur if the regulators are not disabled (e.g. over temp). Although `regulator_force_disable` function call for some consumers can return error numbers, the function is called for all consumers.

## regulator\_bulk\_free

**LINUX**

## Name

`regulator_bulk_free` — free multiple regulator consumers

## Synopsis

```
void regulator_bulk_free (int num_consumers, struct  
regulator_bulk_data * consumers);
```

## Arguments

*num\_consumers*

Number of consumers

*consumers*

Consumer data; clients are stored here.

## Description

This convenience API allows consumers to free multiple regulator clients in a single API call.

# regulator\_notifier\_call\_chain

**LINUX**

## Name

`regulator_notifier_call_chain` — call regulator event notifier

## Synopsis

```
int regulator_notifier_call_chain (struct regulator_dev *  
rdev, unsigned long event, void * data);
```

## Arguments

*rdev*

regulator source

*event*

notifier block

*data*

callback-specific data.

## Description

Called by regulator drivers to notify clients a regulator event has occurred. We also notify regulator clients downstream. Note lock must be held by caller.

# regulator\_mode\_to\_status

**LINUX**

## Name

`regulator_mode_to_status` — convert a regulator mode into a status

## Synopsis

```
int regulator_mode_to_status (unsigned int mode);
```

## Arguments

*mode*

Mode to convert

## Description

Convert a regulator mode into a status.

# regulator\_register

## LINUX

## Name

`regulator_register` — register regulator

## Synopsis

```
struct regulator_dev * regulator_register (struct
regulator_desc * regulator_desc, struct device * dev, const
struct regulator_init_data * init_data, void * driver_data,
struct device_node * of_node);
```

## Arguments

*regulator\_desc*

regulator to register

*dev*

struct device for the regulator

*init\_data*

platform provided init data, passed through by driver

*driver\_data*

private regulator data

*of\_node*

OpenFirmware node to parse for device tree bindings (may be NULL).

## Description

Called by regulator drivers to register a regulator. Returns 0 on success.

## regulator\_unregister

**LINUX**



## Name

`regulator_unregister` — unregister regulator

## Synopsis

```
void regulator_unregister (struct regulator_dev * rdev);
```

## Arguments

*rdev*

regulator to unregister

## Description

Called by regulator drivers to unregister a regulator.

# regulator\_suspend\_prepare

## LINUX

## Name

`regulator_suspend_prepare` — prepare regulators for system wide suspend

## Synopsis

```
int regulator_suspend_prepare (suspend_state_t state);
```

## Arguments

*state*

system suspend state

## Description

Configure each regulator with it's suspend operating parameters for state. This will usually be called by machine suspend code prior to suspending.

# regulator\_suspend\_finish

## LINUX

Kernel Hackers Manual January 2013

## Name

`regulator_suspend_finish` — resume regulators from system wide suspend

## Synopsis

```
int regulator_suspend_finish ( void );
```

## Arguments

*void*

no arguments

## Description

Turn on regulators that might be turned off by `regulator_suspend_prepare` and that should be turned on according to the regulators properties.

# regulator\_has\_full\_constraints

## LINUX

Kernel Hackers Manual January 2013

## Name

`regulator_has_full_constraints` — the system has fully specified constraints

## Synopsis

```
void regulator_has_full_constraints ( void );
```

## Arguments

*void*

no arguments

## Description

Calling this function will cause the regulator API to disable all regulators which have a zero use count and don't have an `always_on` constraint in a `late_initcall`.

The intention is that this will become the default behaviour in a future kernel release so users are encouraged to use this facility now.

# regulator\_use\_dummy\_regulator

## LINUX

Kernel Hackers Manual January 2013

## Name

`regulator_use_dummy_regulator` — Provide a dummy regulator when none is found

## Synopsis

```
void regulator_use_dummy_regulator ( void );
```

## Arguments

*void*

no arguments

## Description

Calling this function will cause the regulator API to provide a dummy regulator to consumers if no physical regulator is found, allowing most consumers to proceed as

though a regulator were configured. This allows systems such as those with software controllable regulators for the CPU core only to be brought up more readily.

## rdev\_get\_drvdata

### LINUX

Kernel Hackers Manual January 2013

### Name

`rdev_get_drvdata` — get rdev regulator driver data

### Synopsis

```
void * rdev_get_drvdata (struct regulator_dev * rdev);
```

### Arguments

*rdev*

regulator

### Description

Get rdev regulator driver private data. This call can be used in the regulator driver context.

# regulator\_get\_drvdata

## LINUX

Kernel Hackers Manual January 2013

### Name

`regulator_get_drvdata` — get regulator driver data

### Synopsis

```
void * regulator_get_drvdata (struct regulator * regulator);
```

### Arguments

*regulator*

regulator

### Description

Get regulator driver private data. This call can be used in the consumer driver context when non API regulator specific functions need to be called.

# regulator\_set\_drvdata

## LINUX

## Name

`regulator_set_drvdata` — set regulator driver data

## Synopsis

```
void regulator_set_drvdata (struct regulator * regulator, void
* data);
```

## Arguments

*regulator*

regulator

*data*

data

## rdev\_get\_id

### LINUX

## Name

`rdev_get_id` — get regulator ID

## Synopsis

```
int rdev_get_id (struct regulator_dev * rdev);
```

## **Arguments**

*rdev*

regulator