wolfson®
microelectronics

A Dynamic Voltage and Current Regulator Control Interface for the Linux Kernel.

Liam Girdwood

wolfson microelectronics

- **Introduction to Regulator Based Systems.**

  – Static & Dynamic System Power

  – Regulator Basics & Power Domains

  – PMP / Internet Tablet Example

- **Kernel Regulator Framework**

  – Consumer Interface

  – Regulator Driver Interface

  – Machine Interface

  – sysfs Interface (ABI)

- **Real World Examples**

  – CPUfreq & CPU Idle

  – LCD Backlight

  – Audio

  – NAND/NOR

- **Resources & Status**

- **Thanks**

- **Q & A**

Introduction to Regulator Based Systems
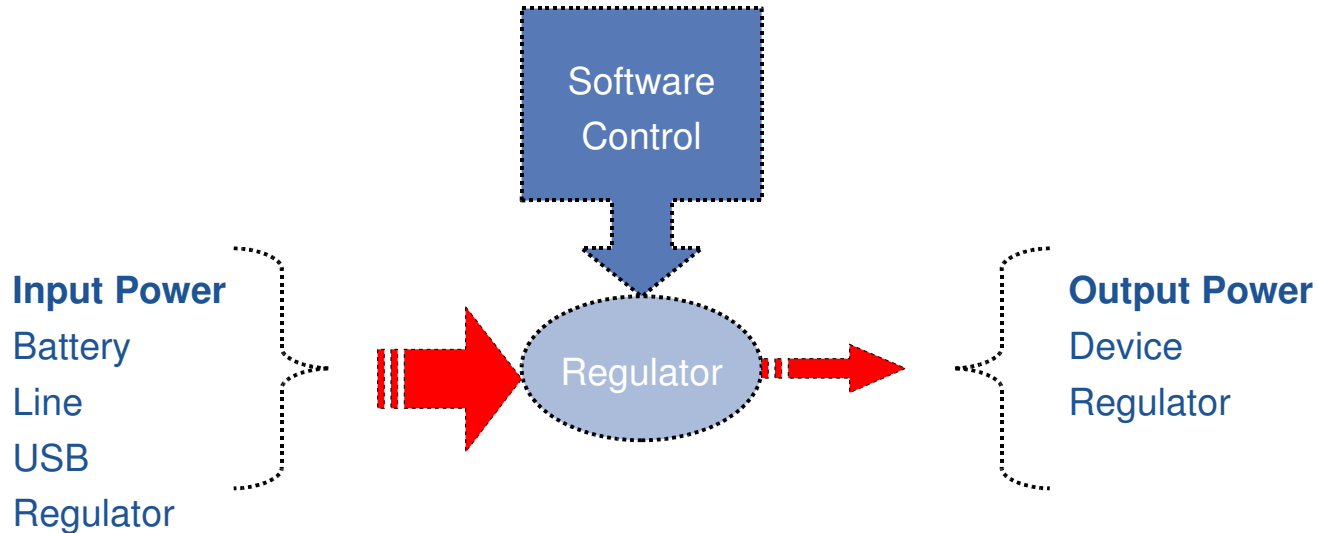
- Semiconductor power consumption has two components – static and dynamic.

$$Power_{(Total)} = P_{(static)} + P_{(dynamic)}$$

- Static power is leakage current.

  – Smaller than dynamic power when system is active.

  – Main power drain in system standby state.

- Dynamic power is active current.

  – Signals switching. (e.g. clocks)

  – Analog circuits changing state (e.g. audio playback).

$$Power_{(dynamic)} = CV^2F$$

- Regulators can be used to save static and dynamic power

wolfson ®
microelectronics

Software Control

Input Power
Battery
Line
USB
Regulator

Regulator
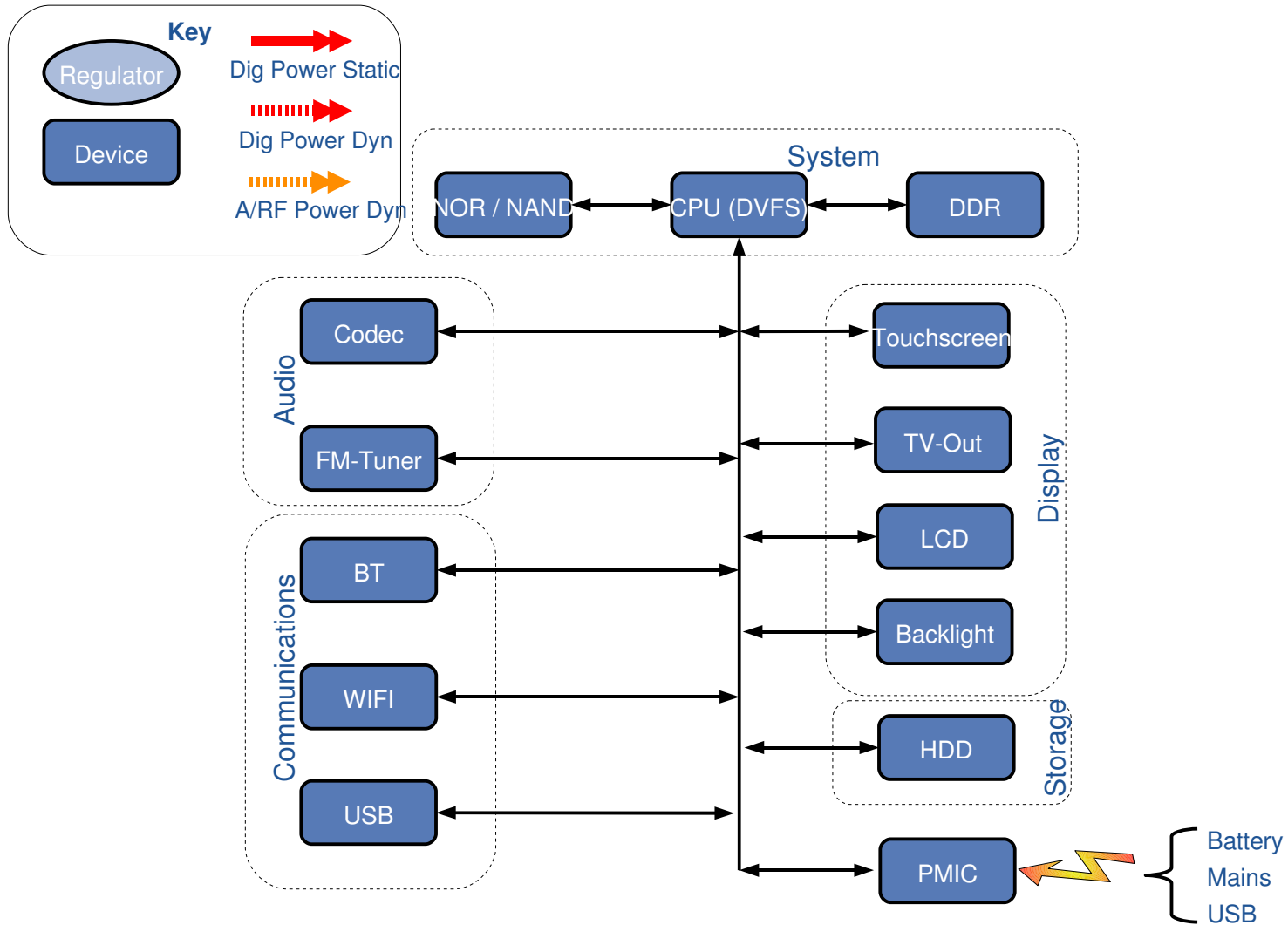
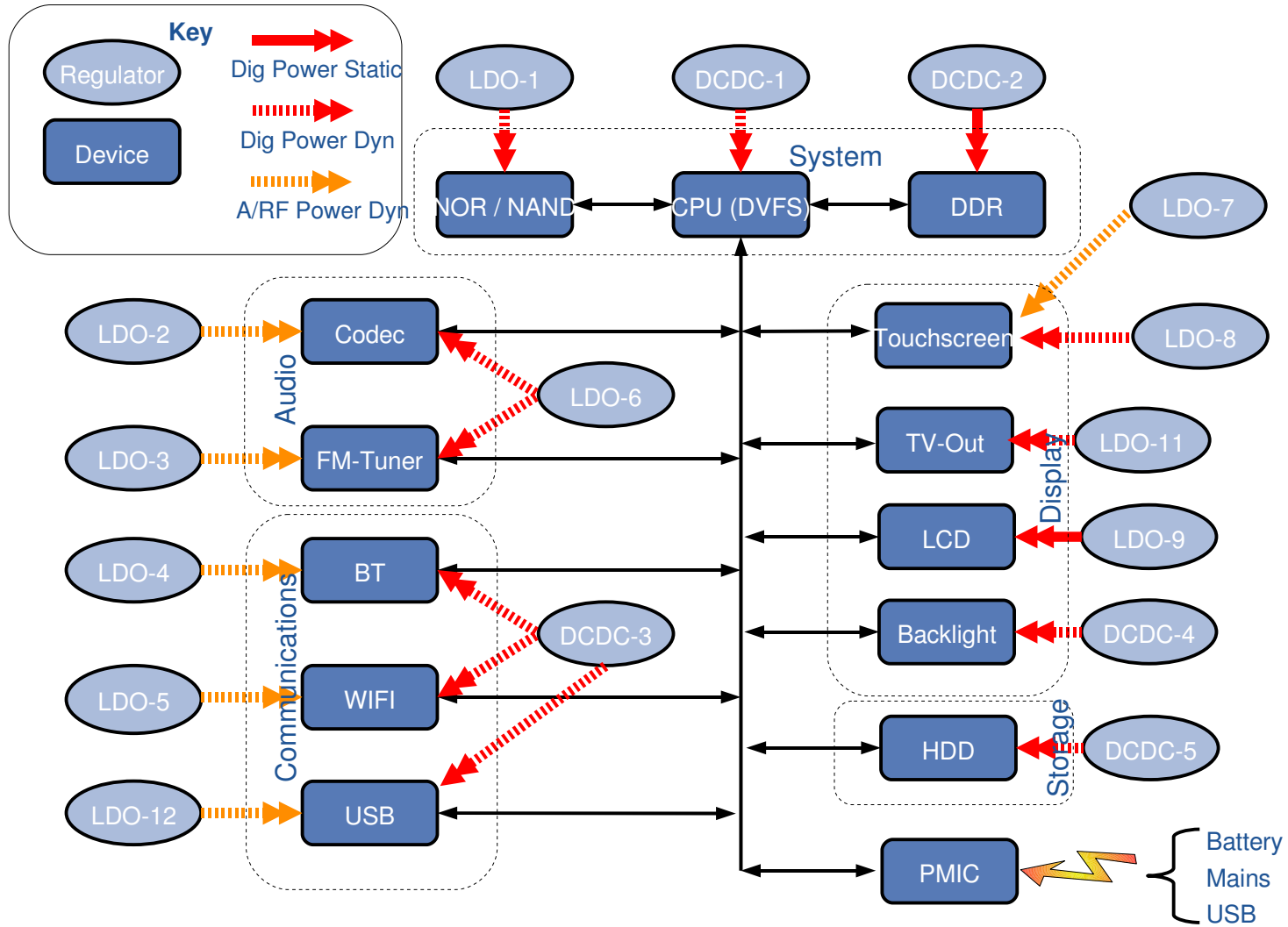**Output Power**
Device
Regulator

- Regulates the output power from input power.

  - Voltage control – *"input is 5V output is 1.8V"*

  - Current limiting – *"limit output current to 20mA max"*

  - Simple switch – *"switch output power on/off"*

wolfson
microelectronics
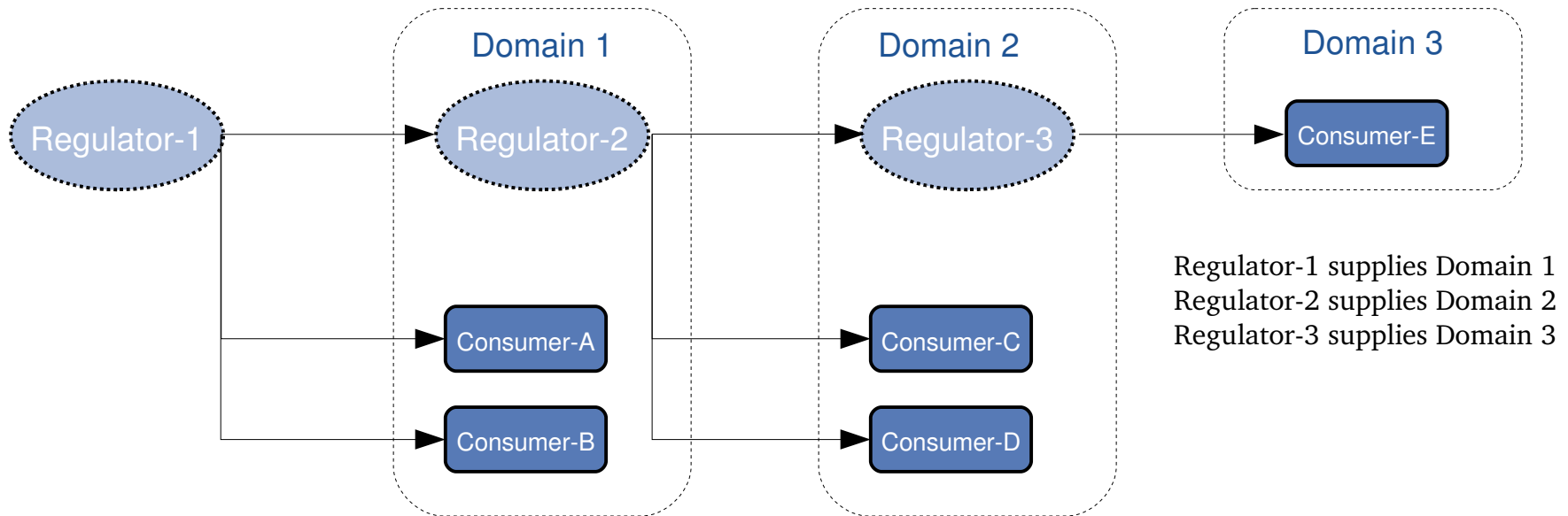
Regulator-1 supplies Domain 1
Regulator-2 supplies Domain 2
Regulator-3 supplies Domain 3

- Power domain supplied power by the **output** of a

  – regulator,

  – switch

  – or by another power domain.

- Has power constraints to protect hardware.

System Architecture (PMP, Internet Tablet)

Key

- Regulator
- Device
- Dig Power Static
- Dig Power Dyn
- A/RF Power Dyn

System

- NOR / NAND
- CPU (DVFS)
- DDR

Audio

- Codec
- FM-Tuner

Communications

- BT
- WIFI
- USB

Display

- Touchscreen
- TV-Out
- LCD
- Backlight

Storage

- HDD

PMIC

- Battery
- Mains
- USB

System Architecture (PMP, Internet Tablet)

Kernel Regulator Framework

- Designed to provide a standard kernel interface to control voltage and current regulators.

- Allow systems to dynamically control regulator power output in order to save power and prolong battery life.

- Applies to both

  – voltage regulators (where voltage output is controllable)

  – current sinks (where current limit is controllable)

- Divided into four separate interfaces.

  – **Consumer** interface for device drivers

  – **Regulator** drivers interface for regulator drivers

  – **Machine** interface for board configuration

  – **sysfs** interface for userspace

- Consumers are client device drivers that use regulator(s) to control their power supply.

- Consumers are constrained by the constraints of the power domain they are on

  – Consumers can't request power settings that may damage themselves, other consumers or the system.

- Classified into two types

  – Static (only need to enable/disable)

  – Dynamic (need to change voltage/ current limit)

- ## Access to regulator is by

  ```
  regulator = regulator_get(dev, "Vcc");
  regulator_put(regulator);
  ```

- ## Enable and disable

  ```
  int regulator_enable(regulator);
  int regulator_disable(regulator);
  int regulator_force_disable(regulator);
  ```

- ## Status

  ```
  int regulator_is_enabled(regulator);
  ```

- Consumers can request their supply voltage with

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

- Constraints are checked before changing voltage.

```
regulator_set_voltage(regulator, 100000, 150000);
```

- Supply voltage can be found with

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

- Consumers can request their supply current limit with

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

- Constraints are checked before changing current limit.

```
regulator_set_current_limit(regulator, 1000, 2000);
```

- Supply current limit can be found with

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

wolfson®
microelectronics

Efficiency (%)

Normal Mode

Idle Mode

Load (mA)
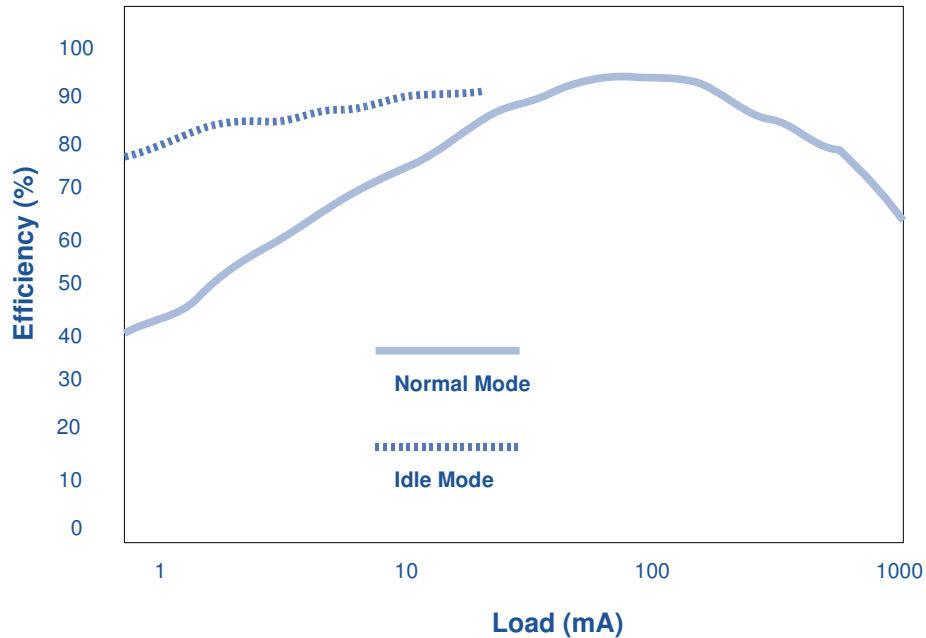
- **Regulators are not 100% efficient.**

- **Efficiency can vary depending on load.**

- **Regulators can change op mode to increase efficiency.**
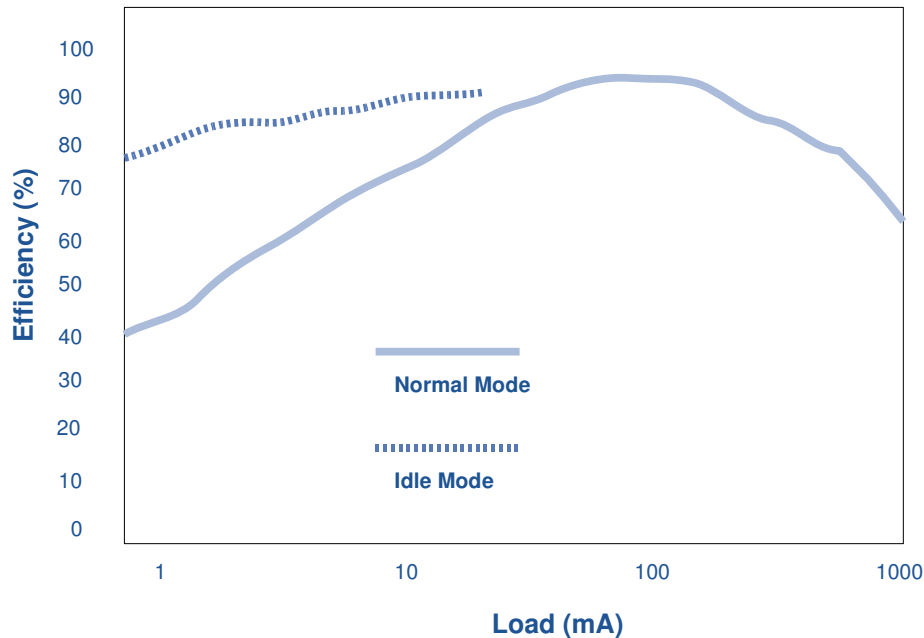
Consumer with 10mA load:-

70% @ Normal = ~13mA

90% @ Idle = ~11mA

Saving ~2mA

We sum total load for regulators > 1 consumer before changing mode.

Optimum efficiency can be requested by calling

```
regulator_set_optimum_mode(regulator, 10000); // 10mA
```

- **Regulator hardware can notify software of certain events.**

  - Regulator failures.

  - Over temperature.

- **Consumers can then handle as required.**

- ## Consumer registration

  `regulator_get(), regulator_put()`

- ## Regulator output power control and status.

  `regulator_enable(), regulator_disable(), regulator_force_disable(),`
  `    regulator_is_enabled()`

- ## Regulator output voltage control and status

  `regulator_set_voltage(), regulator_get_voltage()`

- ## Regulator output current limit control and status

  `regulator_set_current_limit(), regulator_get_current_limit()`

- ## Regulator operating mode control and status

  `regulator_set_mode(), regulator_get_mode(), regulator_set_optimum_mode()`

- ## Regulator events

  `regulator_register_notifier(), regulator_unregister_notifier()`

- Regulator drivers must be registered with the framework before they can be used by consumers.

```
struct regulator_dev *regulator_register(struct regulator_desc
    *regulator_desc, void *reg_data);

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

- Events can be propagated to consumers

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

```c
struct regulator_ops {

/* get/set regulator voltage */
int (*set_voltage) (struct regulator_dev *, int min_uV, int max_uV);
int (*get_voltage) (struct regulator_dev *);

/* get/set regulator current  */
int (*set_current_limit) (struct regulator_dev *,
 int min_uA, int max_uA);
int (*get_current_limit) (struct regulator_dev *);

/* enable/disable regulator */
int (*enable) (struct regulator_dev *);
int (*disable) (struct regulator_dev *);
int (*is_enabled) (struct regulator_dev *);

/* get/set regulator operating mode (defined in regulator.h) */
int (*set_mode) (struct regulator_dev *, unsigned int mode);
unsigned int (*get_mode) (struct regulator_dev *);

/* get most efficient regulator operating mode for load */
unsigned int (*get_optimum_mode) (struct regulator_dev *, int input_uV,
  int output_uV, int load_uA);
};
```

- Regulator drivers can register their services with the core.

  `regulator_register(), regulator_unregister()`

- Regulators can send events to the core and hence to all consumers.

  `regulator_notifier_call_chain()`

- Regulator driver private data.

  `rdev_get_drvdata()`

- Fabric driver that is machine specific and describes

  – Power domains

    "Regulator 1 supplies consumers x,y,z."

  – Power domain suppliers

    "Regulator 1 is supplied by default (Line/Battery/USB)." **OR**

    "Regulator 1 is supplied by regulator 2."

  – Power domain constraints

    "Regulator 1 output must be >= 1.6V and <=1.8V"

Fabric that glues regulators to consumer devices

e.g. NAND is supplied by LDO1

Input Power → LDO1 → **Output Power**
NAND
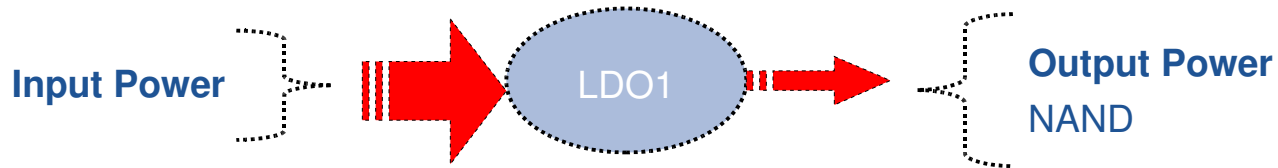
This attaches LDO1 to supply power to the NAND "Vcc" supply pin(s).

```
regulator_set_device_supply ("LDO1", dev, "Vcc");
```

- Defines safe operating limits for power domain.

- Prevents system damage through unsafe consumer requests.

```
struct regulation_constraints audio_avdd = {
    .min_uV = 3300000,
    .max_uV = 3300000,
    .valid_modes_mask = REGULATOR_MODE_NORMAL,
    .apply_uV = 1,
};


regulator_set_platform_constraints(regulator, &audio_avdd);
```
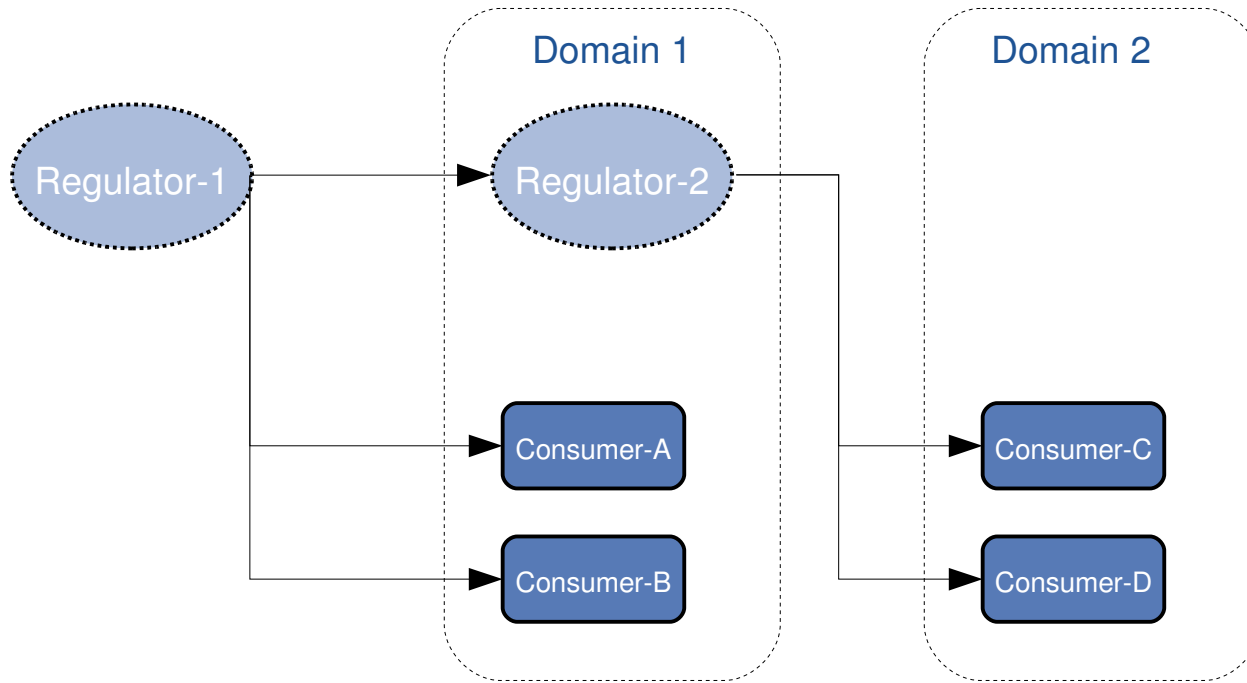
- Some regulators are supplied power by other regulators.

- Ensure regulator 1 is enabled before trying to enable regulator 2.

```
regulator_set_supply("Regulator-2", "Regulator-1");
```

- Regulator --> consumer device mapping

  `regulator_set_device_supply()`

- Regulator -> regulator mapping

  `regulator_set_supply(), regulator_get_supply()`

- Regulator "Power Domain" constraints.

  `regulator_set_platform_constraints()`

- Exports regulator and consumer information to user space

- Is **read** only

  - Voltage

  - Current limit

  - State

  - Operating Mode

  - Constraints

- Could be used to provide more power usage info to powertop

Real World Examples

- CPUfreq scales CPU frequency to meet processing demands

  – Voltage can also be scaled with frequency.

  – Increased with frequency to increase performance/stability.

  – Decreased with frequency to save power.

  ```
  regulator_set_voltage(regulator, 1600000, 1600000); //1.6V
  ```

- CPU Idle can place the CPU in numerous low power idle states.

  – Idle states draw less power and may take advantage of regulator efficiency by changing regulator operating mode.

  ```
  regulator_set_optimum_mode(regulator, 10000); // 10mA
  ```

- LCD back lighting is usually a significant drain of system power.

- Power can be saved by lowering brightness when it's possible to do so.

  - e.g. Some backlights are based on white LED's and can have brightness changed by changing current.

```
regulator_set_current_limit(regulator, 10000, 10000);
```

- Audio hardware consumes requires analog power when there is no audio playback or capture.

- Power could be saved when idle by turning off analog supplies when not in use.

- Power could additionally be saved by turning off components that are not being used in the current use case

  - FM-Tuner could be disabled when MP3's are played.

  - Speaker Amp can be disabled when Headphones are used.
    ```
    regulator_enable(regulator)
    regulator_disable(regulator)
    ```

- NAND & NOR devices consume more power during IO than idle.

- NAND/NOR consumer driver can change regulator operating mode to gain efficiency savings when idle.

```
regulator_set_optimum_mode(regulator, 1000); // 1mA
```

State	Max Load (mA)
Read/Write	35
Erase	40
Erase + rw	55
Idle	1

NAND / NOR chip max load (from datasheet)

- Stable core code working on several machines.

- Currently supports Freescale MC13783, Wolfson WM8350 & WM8400.

- Working with mm kernel by providing patches to the Andrew Morton.

- Project website :- http://opensource.wolfsonmicro.com/node/15

- git - http://opensource.wolfsonmicro.com/cgi-bin/gitweb.cgi

Laura Lawrence

Rob Herring

Nancy Chen

Mark Brown

Greg Kroah-Hartman

Andrew Morton

David Brownell

Sam Ravnborg

Kevin Hillman

Ian Brockbank

Grant More

Questions & Answers