

Porting LTC3589 to a mx5x-family board

List of items that have to be changed when porting to a mx5x board:

- [Regulator Constraints](#) in secondary pmic board file
- [CPU Working Points](#) in secondary pmic board file
- [I2C Communications bus registration](#) in secondary pmic board file
- [PMIC IRQ](#) gpio setup in main board file
- [Included headers](#) in pmic board file
- [Kernel configuration files](#) to properly compile secondary pmic board file and include LTC3589 driver
- [Board specific resistor values](#) in the driver.

Assumptions

The descriptions in this section refer to the task of porting the LTC3589 to another imx5x board. There are some differences between porting to a mx53-based board, vs. another mx5x board (such as the imx51-evk). Those differences are highlighted by sections denoting either:

- **mx53-Specific:**
OR
- **mx5x-Common:**

Please read the next section on [Board File Naming](#) for further information.

The **iMX53-LOCO** board was picked as an example board to port the LTC3589 PMIC to. Since the LTC3589 PMIC is not actually on the iMX53-LOCO it is assumed that the LTC3589 PMIC would be connected in a similar manner as the currently used DA9052 PMIC. Thus:

- DA9052_BUCK_CORE -> SW1
- DA9052_BUCK_MEM -> SW2
- DA9052_BUCK_PERI -> SW3
- DA9052_LD01 -> LD01_STBY
- DA9052_LD02 -> LD02
- DA9052_LD03 -> LD03
- DA9052_LD04 -> LD04

Board File Naming

There are usually 2 board files that represent the mx5x family boards.

mx53-Specific:

1. In this following examples **mx53_loco.c** is the main board file. This file already exists and just has to be modified to remove the reference to the previous pmic and [setting up gpio for the PMIC](#). This is done by deleting the mx53_loco_init_da9052() function call.
2. In this following examples **mx53_loco_pmic_ltc3589.c** is the secondary board file. This

file has to be created. This file will describe the Regulator Constraints, the Consumers who may use the regulator, and it must reference the LTC3589 device driver.

mx5x-Common:

The same modifications are made as the mx53-specific above, but the naming of the files changes.

1. **mx5x_boardname.c**
 - a. Where **x** is the SOC number (mx50/mx51/mx53 etc.)
 - b. Where **boardname** is the name of the board (loco/ard/evk etc.)
2. **mx5x_boardname_pmic_pmicname.c**
 - a. Where **x** is the SOC number (mx50/mx51/mx53 etc.)
 - b. Where **boardname** is the name of the board (loco/ard/evk etc.)
 - c. Where **pmicname** is ltc3589

Included Headers

mx53-Specific:

- **mx53_loco_pmic_ltc3589.c:**

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/platform_device.h>
#include <linux/i2c.h>
#include <linux/err.h>
#include <linux/regulator/ltc3589.h>
#include <linux/regulator/machine.h>
#include <linux/mfd/ltc3589/core.h>
#include <mach/iomux-mx53.h>
#include <mach/irqs.h>
```

mx5x-Common:

The same modifications are made as the mx53-specific above, but the naming of the files changes. Note that some include files may change depending on the mx5x architecture but these files are not related to the pmic implementation.

- **mx5x_boardname_pmic_ltc3589.c:**

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/platform_device.h>
#include <linux/i2c.h>
#include <linux/err.h>
#include <linux/regulator/ltc3589.h>
#include <linux/regulator/machine.h>
#include <linux/mfd/ltc3589/core.h>
#include <mach/iomux-mx5x.h>
#include <mach/irqs.h>
```

Kernel Configuration

1. Add the pmic-board file to the makefile:

mx53-Specific:

arch/arm/mach-mx5/Makefile:

```
...
obj-$(CONFIG_MACH_MX53_EVK) += mx53_evk.o mx53_evk_pmic_mc13892.o
obj-$(CONFIG_MACH_MX53_ARD) += mx53_ard.o mx53_ard_pmic_ltc3589.o
obj-$(CONFIG_MACH_MX53_SMD) += mx53_smd.o mx53_smd_pmic_da9053.o
obj-$(CONFIG_MACH_MX53_LOCO) += mx53_loco.o mx53_loco_pmic_ltc3589.o
...
```

- The secondary pmic board file **mx53_loco_pmic_ltc3589.o** is compiled along with the main board file. Porting to a new board requires changing the name of the secondary pmic board file in the Makefile.

mx5x-Common:

arch/arm/mach-mx5/Makefile:

```
...
obj-$(CONFIG_MACH_MX5X_BOARD) += mx5X_BOARD.o mx5X_BOARD_pmic_ltc3589.o
...
```

- Same as mx53-specific but with a different board name.

2. Select the LTC3589 driver:

- The LTC3589 driver is included in two parts in the Kconfig file under **drivers/mfd/Kconfig**. It can be selected from menuconfig by searching for and selecting the following:
 - **MFD_LTC3589_I2C**
 - **REGULATOR**
 - **REGULATOR_LTC3589** (this depends on the first two)

3. Select the iMX53-LOCO board:

- Since the PMIC-board file is compiled along with the main board file, simply selecting the correct mx5x board from the menuconfig is all that is necessary. For the mx53-LOCO board the parameter to search for is: **MACH_MX53_LOCO**

I2C Bus Registration

The pmic board file contains initialization code needed by the PMIC driver. This code must be run before consumer drivers begin to make requests of the regulator during their initialization. Information on how this is accomplished is in [Appendix B](#).

The I2C bus instance that the PMIC is located on physically (i2c-0/i2c-1/etc.) has to be specified. This is done in the call to `i2c_register_board_info()`.

mx53-Specific:

On the iMX53-LOCO board the PMIC is now on i2c bus 0, instead of i2c bus 1 like on the iMX53-ARD board.

- **mx53_loco_pmic_ltc3589.c:**

```
static __init int mx53_init_i2c(void)
{
    return i2c_register_board_info(0, &lttc3589_i2c_device, 1);
}
```

mx5x-Common:

The same modifications are made as the mx53-specific above, but the naming of the files changes.

- **mx5x_Boardname_pmic_ltc3589.c:**

```
static __init int mx5x_init_i2c(void)
{
    return i2c_register_board_info(bust#, &lttc3589_i2c_device,
1);
}
```

Setup PMIC GPIO/Interrupt Lines

The interrupt lines can be defined for the PMIC, and added to the i2c device structure to allow the ltc3589 driver to initialize a work handler and make a call to request_irq() if needed.

The current version of the ltc3589 driver does not utilize the notification of an undervoltage warning from the LTC3589 so does not actually request an IRQ.

mx53-Specific:

In **mx53_loco.c:**

```
#define LOCO_PMIC_INT          (6*32 + 11)      /* GPIO_7_11 */
#define LOCO_PMIC_RDY          (6*32 + 13)      /* GPIO_7_13 */
#define LOCO_PMIC_PBSTAT       (0*32 + 6) /* GPIO_1_6 */
...
...
...
static struct pad_desc mx53_loco_pads[] = {
    ...
    ...
    ...
    /* PMIC */
    MX53_PAD_GPIO_16__GPIO_7_11,
    MX53_PAD_GPIO_18__GPIO_7_13,
    MX53_PAD_GPIO_6__GPIO_1_6,
};
...
...
```

```

...
static void __init mx53_loco_io_init(void)
{
    ...
    ...
    ...

    /* PMIC */
    gpio_request(LOCO_PMIC_INT, "pmic-int");
    gpio_direction_input(LOCO_PMIC_INT); /*PMIC_INT*/
    gpio_request(LOCO_PMIC_RDY, "pmic-rdy");
    gpio_direction_input(LOCO_PMIC_RDY); /*PMIC_RDY*/
    gpio_request(LOCO_PMIC_PBSTAT, "pmic-pbstat");
    gpio_direction_input(LOCO_PMIC_PBSTAT); /*PMIC_PBSTAT*/
}

```

mx5x-Common:

Similar changes would be made to the mx5x board file to setup GPIO lines.

Board-Specific Resistor Values

mx5x-Common:

If the board is ported to a non-mx53 based design, there are modifications to the driver that are required.

In **ltc3589-regulator.c**:

- A **#ifdef** has to be added to turn off these MX53-specific functions or the code will not compile properly.
- The **BOARD_SPECIFIC_VALUE** resistor values need to be specified according to what is located on the board.

```

static int ltc3589_regulator_probe(struct platform_device *pdev)
{
    ...
    ...
    ...

    #ifdef CONFIG_ARCH_MX53
        if (cpu_is_mx53_rev(CHIP_REV_2_0) >= 1) {
            ltc3589_ldo2_r2 = LTC3589_LDO2_R2_T02;
            ltc3589_sw2_r2 = LTC3589_SW2_R2_T02;
        } else {
            ltc3589_ldo2_r2 = LTC3589_LDO2_R2_T01;
            ltc3589_sw2_r2 = LTC3589_SW2_R2_T01;
        }
    #endif
    ltc3589_ldo2_r2 = BOARD_SPECIFIC_VALUE;
}

```

```

ltc3589_sw2_r2 = BOARD_SPECIFIC_VALUE;
...
...
...

return 0;
}

```

Regulator Constraints

The power constraints of each regulator that the PMIC provides change depending on the board. The constraints placed in the pmic board file dictate the actual parameters of the regulators on a particular board including voltage, current, or mode changes. In the corresponding **LTC3589-regulator.c** driver, operations that are possible for the pmic (on all boards) are defined. Functions are implemented for actually performing these operations as well (via the i2c communications bus). For an explanation of the capabilities of each regulator on the LTC3589 refer to the **Study of the LTC3589 Driver** document.

For instance, devices connected to a regulator will have a range of voltages (min-max) that they can safely allow. Each regulator for a particular board has to know what the range of allowable voltages are to prevent accidental requests for harmful voltage settings.

mx53-Specific:

mx53_loco_pmic_ltc3589.c:

```

static struct regulator_init_data sw3_init = {
    .constraints = {
        .name = "SW3",
        .min_uV = 1342000,
        .max_uV = 2775000,
        .valid_ops_mask = REGULATOR_CHANGE_VOLTAGE,
        .always_on = 1,
        .boot_on = 1,
        .initial_state = PM_SUSPEND_MEM,
        .state_mem = {
            .uV = 2500000,
            .mode = REGULATOR_MODE_NORMAL,
            .enabled = 1,
        },
        .state_standby = {
            .uV = 2000000,
            .mode = REGULATOR_MODE_NORMAL,
            .enabled = 1,
        },
    },
};

```

- If the regulator allows stepping up/down the voltage then the constraint items that are important to modify are:
 - **min_uV**: Minimum voltage setting for this regulator.
 - **max_uV**: Maximum voltage setting for this regulator.
 - **valid_ops_mask**: Set to **REGULATOR_CHANGE_VOLTAGE** to permit voltage changes. This setting can be OR'd | with other allowed operations.
- If the regulator is capable of reducing its voltage during standby (like the SWx regulators on the LTC3589) then that standby voltage has to be set. During the **Suspend_Prep** phase, a PM message is sent by the power management system in linux, to the regulators that support standby levels. The regulators must prepare their system to be ready for the VSTBY signal. In the case of the LTC3589 this requires setting its xxBTv2 registers. There are 3 PM levels supported in Linux: **state_mem**, **state_standby**, and **state_disk** but **state_mem** is currently the most widely used. State_mem consists of the following items:
 - **uV**: Suspend voltage the PMIC should apply to the regulator.
 - **mode**: Regulator operating mode. Most drivers use normal regulator power supply mode.
 - **enabled**: Is the regulator enabled when in this suspend state?

mx53_loco_pmic_ltc3589.c:

```
static struct regulator_init_data sw4_init = {
    .constraints = {
        .name = "SW4",
        .valid_modes_mask = REGULATOR_MODE_NORMAL |
                           REGULATOR_MODE_STANDBY,
        .valid_ops_mask = REGULATOR_CHANGE_MODE |
                           REGULATOR_CHANGE_STATUS,
        .apply_uV = 1,
        .boot_on = 1,
    }
};
```

- If the regulator allows changing modes (burst mode/ active / idle / standby) then the following items that apply are:
 - **valid_ops_mask**: Set to **REGULATOR_CHANGE_MODE** to permit mode changes.
 - **valid_modes_mask**: Set to all modes that are supported. For instance on SWx this could be set to **REGULATOR_MODE_FAST | REGULATOR_MODE_NORMAL | REGULATOR_MODE_STANDBY**

mx53_loco_pmic_ltc3589.c:

```
static struct regulator_init_data ldo4_init = {
    .constraints = {
        .name = "LDO4",
```

```

        .min_uV = 1725000,
        .max_uV = 3300000,
        .valid_ops_mask = REGULATOR_CHANGE_VOLTAGE,
        .valid_modes_mask = REGULATOR_MODE_NORMAL,
        .always_on = 1,
        .boot_on = 1,
    },
};

drivers/regulator/ltc3589-regulator.c:
static struct regulator_desc ltc3589_reg[] = {
    ...
    ...
    {
        .name = "LDO4",
        .id = LTC3589_LDO4,
        .ops = &ltc3589_ldo4_ops,
        .type = REGULATOR_VOLTAGE,
        .n_voltages = ARRAY_SIZE(LDO4_VSEL_table),
        .owner = THIS_MODULE,
    },
};

mx53_loco.c:
static struct tve_platform_data tve_data = {
    .dac_reg = "LDO4",
};

drivers/video/mxc/tve.c:
static int tve_probe(struct platform_device *pdev)
{
    ...
    ...
    ...
    tve.dac_reg = regulator_get(&pdev->dev, plat_data->dac_reg);
    if (!IS_ERR(tve.dac_reg)) {
        regulator_set_voltage(tve.dac_reg, 2750000, 2750000);
        regulator_enable(tve.dac_reg);
    }
}

```

- Each device that draws power from a regulator is considered a consumer. Here the TVE device is the Consumer and the Producer is regulator “LDO4”. Additional information on how to define the Consumer/Producer relationship is detailed in the section on [Regulator API](#).

mx5x-Common:

This section also applies to porting to another mx5X family board. All that is different is the name of the files modified:

- **mx5x_***boardname*_pmic_ltc3589.c
- **mx5x_***boardname*.c

CPU Working Points

CPU Working Points describe a set of clock frequency + voltage settings. Once multiple working points are defined, the power management system move between working points by asking the cpu to reduce its frequency using the **cpu_freq API** and the pmic to reduce cpu voltage via the [Regulator API](#).

mx53-Specific:

The **cpu_wp** structure has been implemented already for all mx53 soc types. The methods for getting/setting the wp are also implemented. Everything is defined in the files **arch/arm/mach-mx5/mx53_wp.c/.h**.

The type of mx53 SOC (535/537/538/etc) is determined automatically by checking the core clock speed in **arch/arm/mach-mx5/clock.c**.

mx5x-Common:

The **cpu_wp** structure and get/set methods for the mx5x boards is defined in the board file. For instance in **arch/arm/mach-mx5/mx51_babbage.c** there are 3 working points defined like so:

```
static struct cpu_wp cpu_wp_auto[] = {
{
    .pll_rate = 1000000000,
    .cpu_rate = 1000000000,
    .pdf = 0,
    .mfi = 10,
    .mfd = 11,
    .mfn = 5,
    .cpu_podf = 0,
    .cpu_voltage = 1175000,},
{
    .pll_rate = 800000000,
    .cpu_rate = 800000000,
    .pdf = 0,
    .mfi = 8,
    .mfd = 2,
    .mfn = 1,
    .cpu_podf = 0,
    .cpu_voltage = 1100000,},
{
    .pll_rate = 800000000,
    .cpu_rate = 166250000,
    .pdf = 4,
    .mfi = 8,
    .mfd = 2,
    .mfn = 1,
    .cpu_podf = 4,
```

```
        .cpu_voltage = 850000,},  
};
```

I2C interface

The LTC3589 driver communicates with the PMIC via I2C. It contains a driver that allows reading/writing to its registers. This driver (**ltc3589-i2c.c**) provides a communications interface for the rest of the ltc3589 driver.

Regulator API

The Regulator API is a generic API for representing a regulator in Linux. The job of a regulator is to register its constraints and handle incoming requests for adjusting the voltage/current of a particular regulator. The **ltc3589-regulator.c** driver contains functions for setting the voltage, modes, and enabling/disabling each regulator. It also links these functions to a structure describing what operations are permitted by the Linux Power Management interface on the regulators.

The Consumers (various device drivers that consume power and want to request voltage/current changes of the regulator feeding them) can get a handle to their regulator by calling **regulator_get()** and passing in the device name and id.

The id can be either:

- The name of the 'supply'
- The name of the regulator

Supply example

'Supply' name is defined in **arch/arm/mach-mx5/mx51_babbage_pmic_mc13892.c**

```
static struct regulator_consumer_supply vvideo_consumers[] = {  
    {  
        /* sgtl5000 */  
        .supply = "VDDIO",  
        .dev_name = "1-000a",  
    },  
};
```

```

};

...
...
...
static struct regulator_init_data vvideo_init = {
    .constraints = {
        .name = "VVIDEO",
        .min_uV = mV_to_uV(2775),
        .max_uV = mV_to_uV(2775),
        .valid_ops_mask = REGULATOR_CHANGE_VOLTAGE |
            REGULATOR_CHANGE_STATUS,
        .apply_uV = 1,
    },
    .num_consumer_supplies = ARRAY_SIZE(vvideo_consumers),
    .consumer_supplies = vvideo_consumers,
};

```

'Supply' name is referenced by the consumer driver **sound/soc/codecs/sgtl5000.c**

```
reg = regulator_get(&client->dev, "VDDIO");
```

Regulator example

Regulator name is defined in **drivers/regulator/ltc3589-regulator.c**:

```

static struct regulator_desc ltc3589_reg[] = {
    ...
    ...
    ...
    {
        .name = "SW2",
        .id = LTC3589_SW2,
        .ops = &lttc3589_sw_ops,
        .type = REGULATOR_VOLTAGE,
        .n_voltages = 0x1F + 1,
        .owner = THIS_MODULE,
    },
};

```

and passed to the bus_freq driver via platform data in **arch/arm/mach-mx5/mx53_ard.c**

```

static struct mxc_bus_freq_platform_data bus_freq_data = {
    .gp_reg_id = "SW1",
    .lp_reg_id = "SW2",
};

```

Regulator name is referenced by the consumer driver **arch/arm/mach-mx5/bus_freq.c**

```
lp_reg_id = p_bus_freq_data->lp_reg_id;  
lp_regulator = regulator_get(NULL, lp_reg_id);
```

Appendix A

Referenced files for porting LTC3589 to the imx53-loco board.

Added

- arch/arm/mach-mx5/
 - mx53_loco_pmic_ltc3589.c

Modified

- arch/arm/mach-mx5/
 - mx53_loco.c

Important (relevant files from ARD board)

- arch/arm/mach-mx5/
 - mx53_wp.c/.h
 - mx53_ard.c
 - mx53_ard_pmic_ltc3589.c
- drivers/regulator/
 - ltc3589-regulator.c
- drivers/mfd/
 - ltc3589-i2c.c

Appendix B

Initialization Phases

Phase 4:

The regulator constraints from the pmic board file are registered with the Linux I2C bus framework with a **i2c_register_board_info()** call using **subsys_initcall()**. The corresponding **ltc3589-i2c.c** driver also uses **subsys_initcall()** to add its probing function to the Linux I2C bus framework. The driver's probe function is thus launched, and the init function that was passed from the pmic board file is run **mx53_ltc3589_init()**. This init function registers each regulator's constraints with the ltc3589-regulator module using the **ltc3589_register_regulator()** function. This regulator registration function creates 8 platform device instances.

Phase 4s:

The ltc3589-regulator.c module registers itself as a platform driver using the **subsys_initcall_sync()** macro. This links the regulator driver with the regulator constraints and the regulator driver probe function is run **ltc3589_regulator_probe**. The probe function makes a call to **regulator_register()** which is how the regulators are actually registered with the Linux Regulator API.

Phase 6/6s:

Device drivers are initialized during this phase. These are often the device drivers for devices that act as Consumers of regulator power.

Phase 7:

The function **ltc3589_pmic_init()** in the pmic board file is run. This function enables all the regulators using the **regulator_enable()** function of the Linux Regulator API.