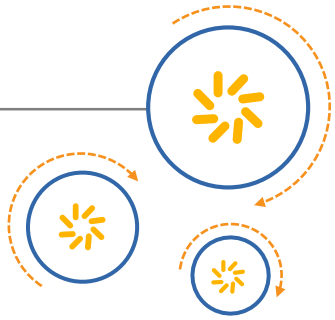
Qualcomm Technologies, Inc.



APSS GPIO Pin Control Software

80-NL239-47 E

November 26, 2015

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Revision history

Revision	Date	Description
A	June 2014	Initial release
B	October 2014	Updated the following to include MSM8936/MSM8939 and MSM8909 information: ▪ Document title Sections 1.1, 1.2, and 2.1
C	April 2015	Updated the document title; updated Sections 1.1 and 2.1
D	August 2015	Updated Sections 1.1 and 2.1
E	November 2015	Added Chapter 6; added Sections 2.1.2, 2.1.3, and 2.1.4; updated Sections 1.1 and 2.1

Contents

Figures

1 Introduction

1.1 Purpose

This document explains how to migrate a Linux device driver from GPIO lib implementations to the pin control framework to configure SoC pin resources for respective devices. It describes different use cases for SoC pins and the corresponding migration strategy on the MSM8916, MSM8936/MSM8939, MSM8909, MSM8937, MSM8952, MSM8953, MSM8956/MSM8976, and MDM9x07 chipsets.

This document is intended for Linux device driver developers. It is intended for software engineers who want to configure GPIOs in the APPS subsystem. This document does not comprehensively explain the pin control framework itself. It assumes that the readers have a basic knowledge of the framework and the underlying driver implementation. This document is applicable to the MSM8916, MSM8936/MSM8939, MSM8909, MSM8937, MSM8952, MSM8953, MSM8956/MSM8976, and MDM9x07 chipsets.

1.2 Conventions

Function declarations, function names, type declarations, attributes, and code samples appear in a different font, for example, `#include`.

Code variables appear in angle brackets, for example, `<number>`.

If you are viewing this document using a color monitor, or if you print this document to a color printer, **red boldface** indicates code that is to be **added**, and ~~blue strikethrough~~ indicates code that is to be replaced or removed.

Shading indicates content that has been added or changed in this revision of the document.

1.3 Technical assistance

For assistance or clarification on information in this document, submit a case to Qualcomm Technologies, Inc. (QTI) at https://createpoint.qti.qualcomm.com/.

If you do not have access to the CDMATech Support website, register for access or send email to support.cdmatech@qti.qualcomm.com.

2 Linux pin control overview

The Linux pin control framework is designed specifically for embedded applications on SoCs where the outfacing pins are considered precious and limited resources. A pinmuxing logic allows a pin or a group of pins to be reused for different purposes. Each use case of these pins require different attributes to be configured to drive the signal on the pins. The pin control and pinmux framework allow users to specify different configuration attributes as well as define multiple uses/functions for a group of pins.

2.1 Hardware overview

There are 122 GPIOs in the MSM8916 and MSM8936/MSM8939 chipsets, 112 GPIOs in MSM8909, 134 GPIOs in MSM8952 and MSM8937, 80 GPIOs in MDM9x07, 142 GPIOs in MSM8953, and 145 GPIOs in MSM8956/MSM8976 chipset. This section provides an overview of the hardware aspect of the GPIOs.

2.1.1 GPIO pad structure

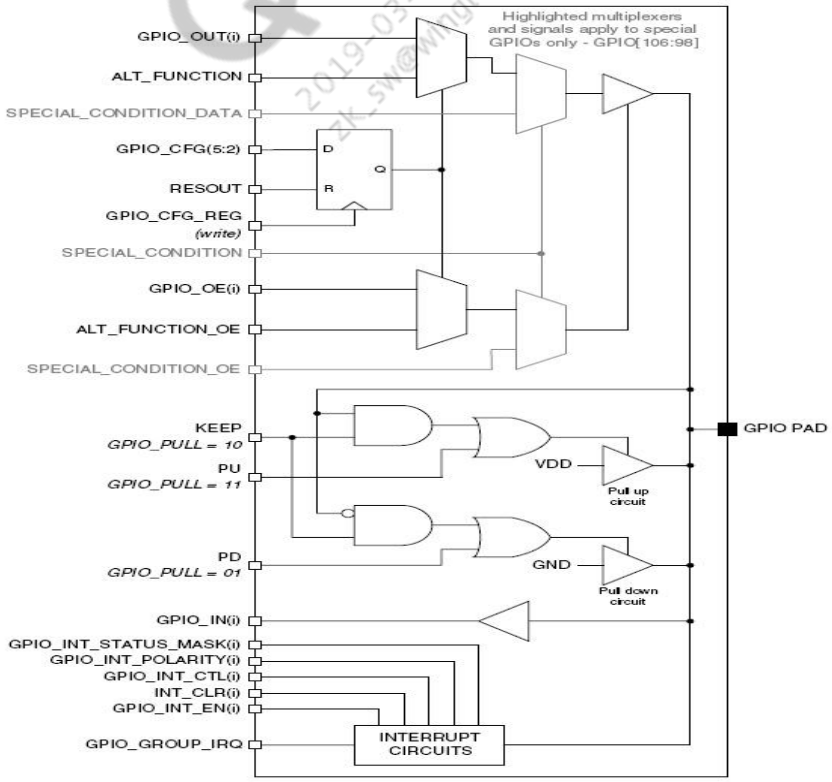


Figure 2-1 Conceptual structure diagram of GPIO pad

As shown in Figure 2-1, a GPIO pin can be configured as general input, output, or one of the several alternate functions. When used as an input, a GPIO pin can also be set up as an interrupt source, and a subset of GPIOs can generate interrupts that can bring the MSM™ out of deep sleep (XO shutdown or VDD minimization). GPIOs can also be configured to have internal pull circuits (up, down, or no-pull) and different driving strengths.

2.1.2 GPIO registers – Configuration

NOTE: This section was added to this document revision.

The configuration of a GPIO is controlled by GPIO_CFGn, n = 0 ... 121 registers. Each register controls the configuration of one GPIO. Table 2-1 lists the bit field for the GPIO_CFGn register.

Table 2-1 GPIO_CFGn register bit field

Bits	Name	Description
10	GPIO_HIHYS_EN	Controls the hysteresis of GPIO[n] on input modes
9	GPIO_OE	Controls the OE for GPIO[n] when it is in GPIO mode
8:6	DRV_STRENGTH	Controls the GPIO pad drive strength; this applies regardless of the FUNC_SEL field selection ▪ 000 – 2 mA ▪ 001 – 4 mA ▪ 010 – 6 mA ▪ 011 – 8 mA ▪ 100 – 10 mA ▪ 101 – 12 mA ▪ 110 – 14 mA ▪ 111 – 16 mA
5:2	FUNC_SEL	Many of the GPIO pads have one or more functional hardware interfaces. This field controls how the pad is used. Set this field to 0 for GPIO mode.
1:0	GPIO_PULL	The pad can be configured to employ an internal weak pull-up, pull-down, keeper, or no-pull function; this applies regardless of the FUNC_SEL field selection. ▪ 00 – No-pull ▪ 01 – Pull-down (PD) ▪ 10 – Keeper ▪ 11 – Pull-up (PU)

2.1.3 GPIO registers – Input and output

NOTE: This section was added to this document revision.

When the FUNC_SEL of a GPIO is set to 0, the GPIO is used as a general purpose pin, and it is used as either an input or output pin. This is controlled by the GPIO_OE bit of the GPIO_CFGn register. When this bit is set to 1, it is an output pin, and if the bit is set to 0, it is becomes an input pin.

When set as an output pin, the GPIO state is controlled by GPIO_IN_OUTn, n = 0 ... 121 registers. When set as an input pin, the GPIO state can be read from the same register. Table 2-2 lists the bit field of the GPIO_IN_OUTn register.

Table 2-2 GPIO_IN_OUT register bit field

Bits	Name	Description
31:2	Reserved	Reserved field
1	GPIO_OUT	Controls the output state of an output pin
0	GPIO_IN	Allows the state of an input pin to be read

2.1.4 GPIO registers – Interrupt

NOTE: This section was added to this document revision.

All GPIOs are set up as interrupt sources. GPIO interrupt is controlled by the GPIO_INTR_CFGn, n = 0 ... 121 register. Table 2-3 lists the bit field of the GPIO_INTR_CFGn register.

Table 2-3 GPIO_INTR_CFGn bit field

Bit	Name	Description
31:9	Reserved	Reserved field
8	DIR_CONN_EN	Tells the Top Level Module Multiplex (TLMM) that GPIO[n] is being used as a direct connect interrupt. ▪ 0 – Disables the GPIO as a direct connect interrupt ▪ 1 – Enables the GPIO as a direct connect interrupt

Bit	Name	Description
7:5	TAEGET_PROC	Determines to which processor a summary interrupt from GPIO[n] gets routed ▪ 0x0: WCSS – Routes the GPIO[n] signal to the WCSS summary interrupt ▪ 0x1: Sensors – Routes the GPIO[n] signal to the sensors summary interrupt ▪ 0x2: LPA_DSP – Routes the GPIO[n] signal to the LPASS summary interrupt ▪ 0x3: RPM – Routes the GPIO[n] signal to the RPM summary interrupt ▪ 0x4: KPSS – Routes the GPIO[n] signal to the KPSS summary interrupt ▪ 0x5: MSS – Routes the GPIO[n] signal to the MSS summary interrupt ▪ 0x6: TZ – Routes the GPIO[n] signal to the TZ summary interrupt ▪ 0x7: None – Does not route to any processor subsystem; this is the default setting
4	INTR_RAW_STATUS_EN	Enables the RAW status for the summary interrupt on this GPIO; this is a power-saving mechanism and should be left disabled unless necessary ▪ 1 – Enable ▪ 0 – Disable
3:2	INTR_DECT_CTL	Controls the edge or level detection of the interrupt controller ▪ 0x0: LEVEL – Level sensitive ▪ 0x1: POS_EDGE – Positive edge sensitive ▪ 0x2: NEG_EDGE – Negative edge sensitive ▪ 0x3: DUAL_EDGE – Sensitive to both edges
1	INTR_POL_CTL	Controls the polarity detection of the interrupt controller ▪ Polarity 1 corresponds to active high ▪ Polarity 0 corresponds to active low ▫ 0 – POLARITY_0 ▫ 1 – POLATRITY_1
0	INTR_ENABLE	Controls the generation of summary interrupt ▪ 1 – Enable ▪ 0 – Disable

The trigger type of the interrupt is controlled by INTR_POL_CTL and INTR_DETC_CTL. Setting/clearing the INTR_ENABLE field enables/disables the corresponding GPIO interrupt.

The interrupt status for GPIO[n] is recorded in the GPIO_INTR_STATUSn, n = 0 ... 121 register. When read, this register returns the interrupt status of the corresponding GPIO.

- The interrupt is active when it returns 1. To clear the GPIO interrupt status, write a 0 into this register.
- The interrupt is not active when it returns 0. To set the interrupt, write a 1 into this register.

2.2 Pin configuration vs pin function

Pin configuration programming is an attribute that governs the nature of a signal to be driven on the lines. Typical configurations include drive strength, pull values, direction, etc. Pin function can be defined as the use of the signal to be driven on the pins.

2.3 Pin states

The framework implementation employs the concept of pin states. A state comprises of configurations and functions to be applied on a group of pins. An owner of one or more pins must furnish the information that comprises the state of a pin group. The framework allows competing owners, i.e., independent clients that want to use the same pins for different purposes, to furnish the respective pin state information.

There is no limit to the number of states an owner can define. However, owners typically define the following three states:

- Default
- Active
- Sleep

For a detailed description of pin control framework and client APIs, see *PINCTRL (Pin Control) Subsystem* (https://www.kernel.org/doc/Documentation/pinctrl.txt).

3 MSM TLMM

MSM SoCs have a pinmux controller known as Top-Level Mode Multiplexer (TLMM). The current version of TLMM is Ver 4.

3.1 Pin types on TLMM

TLMM Ver 4 supports multiplexing and configuration of different types of pins. Any SoC that incorporates TLMM Ver 4 controller can only have pins of types supported by TLMM Ver 4. These types have different register semantics, and in some cases even have their own protocol-specific pin attributes, i.e., attributes different than drive strength, pull, direction, etc. In the context of pin control and based on existing use cases, this document applies to the following two-pin types supported by TLMM Ver 4:

- General-purpose pins – Pins that can be used for multiple purposes
- SDC pins – Pins that are dedicated to SDHC

3.2 Previous software model for TLMM programming

Earlier, the GPIO lib was used to configure pins and apply specific functions to them.

- The pin configurations were defined in the board -<soc>- gpiomux.c file. These configurations were installed during kernel initialization.
- Clients defined at least two sets of configurations, Active and Sleep states.
- The gpio_request() and gpio_free() APIs were used to install and revert the Active and Sleep pin configuration states at runtime.

In some cases, a separate API was provided to the clients to program more than two configurations.

4 Pin control software model

A pin control-based software model differs in the following ways:

- All pin configurations and TLMM information is defined in an SoC-specific file, <soc>--pinctrl.dtsi. This file contains the pin groupings as well as the configurations and functions to be applied to them.
- Individual client device tree nodes reference the configuration nodes defined in <soc>-pinctrl.dtsi. It consists the state of pins. A client can have any number of configuration nodes in a state, and any number of states.
- The state information is parsed with a call to devm_pinctrl_get().
- A client can then choose to install a given state using pinctrl_lookup_state() and pinctrl_select_state().

In the following example, arch/arm/boot/dts/msm8916-pinctrl.dtsi is a sample TLMM pinmux node containing the pin types:

```
&soc {
   tlmm_pinmux: pinctrl@1000000 {
        compatible = "qcom,msm-tlmm-v4";
        reg = <0x1000000 0x300000>;
        interrupts = <0 208 0>;

        /*General purpose pins*/
        gp: gp {
             qcom,pin-type-gp;
             qcom,num-pins = <122>;
             #qcom,pin-cells = <1>;
             msm_gpio: msm_gpio {
                  compatible = "qcom,msm-tlmmv4-gp-intc";
                  gpio-controller;
                  #gpio-cells = <2>;
                  interrupt-controller;
                  #interrupt-cells = <2>;
                  num_irqs = <122>;
             };
        };
        /* SDC pin type */
        sdc: sdc {
```

```
                qcom,pin-type-sdc;
                /* 0-2 for sdc1 4-6 for sdc2 */
                qcom,num-pins = <7>;
                /* Order of pins */
                /* SDC1: CLK -> 0, CMD -> 1, DATA -> 2 */
                /* SDC2: CLK -> 4, CMD -> 5, DATA -> 6 */
                #qcom,pin-cells = <1>;
        };
};
```

5 Software modification

The existing usage of pins on SoCs can be divided into the following three types:

- Use case 1 – Pins driven by controllers, e.g., UART, SPI, and I2C.
- Use case 2 – Pins used by software for bit banging; typically the devices attached externally to the board, e.g., display panels.
- Use case 3 – Pins that are used as a source of interrupts; typically attached externally to the board, e.g., Ethernet.

In some instances, OEMs use the same group of pins for use cases 1 and 2. In such instances, these two cases are modeled as extra states in the device tree.

WARNING: The pinctrl framework places limitations on the context in which pinctrl APIs can be invoked. If the driver violates the context constraints, redesign the corresponding portion. For an explanation of constraints, see *PINCTRL (Pin Control) Subsystem* (https://www.kernel.org/doc/Documentation/pinctrl.txt).

5.1 Use case 1 – Pins driven by controllers

5.1.1 Configuring general-purpose pins only once on bootup

To configure the general-purpose pins only once on bootup:

1. Add a pin grouping node to the msm8916-pinctrl.dtsi file. This step is a continuation of the example described in Chapter 4.

```
&soc {
tlmm_pinmux: pinctrl@1000000 {
…..
….
..
Client1_pins {
/* Uses general purpose pins */
qcom,pins = <&gp 0>, <&gp 1>;
qcom,num-grp-pins = <2>;
/* function setting as specified in board-<soc>-gpiomux.c */
qcom,pin-func = <1>;
label = “client1-bus”;
/* Active configuration of bus pins */
Client1_default: client1_default {
```

```
/* Property names as specified in
• pinctrl-bindings.txt /
drive-strength = <8>; /* 8 MA */
bias-disable; /* No PULL */
};
};
```

2. Modify the client node in the device tree (msm8916.dtsi or msm8916-<board>.dtsi).

```
Soc {
…….
…
..
Client1 {
……
…
/*Note the the default state is programmed by the kernel at the time of
kernel bootup. No driver changes are necessary since at probe time the
default state would already be programmed in TLMM */
pinctrl-states = “default”;
/* Use phandle referrence to default configuration node */
pinctrl-0 = <&Client1_default>;
};
```

5.1.2 Configuring general-purpose pins for Active and Sleep states (runtime)

In this case, a controller driver has Active state for pins (during transactions or default) and Sleep state when performing system suspend. The driver must install the correct state accordingly.

To configure general-purpose pins for Active and Sleep states:

1. Define the pin group node containing Active and Suspend states.

```
&soc {
tlmm_pinmux: pinctrl@1000000 {
…..
….
Client2_pins {
/* Pin group node */
/* Uses general purpose pins */
qcom,pins = <&gp 0>, <&gp 1>;
qcom,num-grp-pins = <2>;
/* function setting as specified in board-<soc>-gpiomux.c */
qcom,pin-func = <1>;
```

```
label = “client2-bus”;
/* Active configuration of bus pins */
Client2_active: client2_active {
/* Configuration node:
• Property names as specified in
• pinctrl-bindings.txt /
drive-strength = <8>; /* 8 MA */
bias-pull-up; /* PULL UP */
};
Client2_suspend: client2_suspend {
/* Configuration node:
• Property names as specified in
• pinctrl-bindings.txt /
drive-strength = <2>; /* 2 MA */
bias-disable; /* No PULL */
};
};
```

2. Modify the client node to include pinctrl states. Delete any explicit references to the GPIOs using the GPIO chip node.

```
Soc {
…….
…
Client2 {
……
pinctrl-states = “active”, “suspend”;
/* Use phandle referrence to active and sleep configurations to
Define the active and sleep pin states */
pinctrl-0 = <&Client2_active>;
pinctrl-1 = <&Client2_sleep>;
client2-gpio-1 = <&msmgpio 0 0>;
client2-gpio-2 = <&msmgpio 1 0>;
};
```

3. Modify the driver to the pinctrl interface.

```
Static int msm_client2_probe (struct platform_device *pdev) {
Struct pinctrl_state *set_state;
Struct client2_data *client2_dd;
…..
…
..
```

```
/* Try to obtain pinctrl handle */
Pdev->dev->pins->p = devm_pinctrl_get(pdev);
/* Lookup the active configuration */ 34
set_state = pinctrl_lookup_state(pdev->dev->pins->p, “active”);
if (!set_state)
goto fail;
else
/*Actually write the active configuration to hardware */
ret = pinctrl_select_state(pdev->dev->pins->p, set_state);

/* Install sleep configuration in runtime suspend function */
Static int client2_suspend( struct platfom_device) {
/* Configure pins to sleep state state */
Struct pinctrl_state *set_state;
/* Actually write the sleep configuration to hardware */
set_state = pinctrl_lookup_state(pdev->dev->pins->p, “sleep”);
if (!set_state)
goto fail;
else
ret = pinctrl_select_state(pdev->dev->pins->p, set_state);
}
```

5.1.3 Pin groups with different function settings in configurations (runtime)

The function applied on a group of pins remains unchanged for the use cases discussed till now. However, in some use cases, the function applied to a pin group changes as part of installing a new state. In these cases, the pin group is modelled as two separate pin groups using the same pins; the driver chooses to install a new state for the pin groups with different/same configurations, however, with different pin functions. The following example shows different pin functions applied to the same pin group:

```
&soc {
tlmm_pinmux: pinctrl@1000000 {
...
..
Pmx_wcnss_5wire_active {
qcom,pins = <&gp 40>, <&gp 41>, <&gp 42>, <&gp 43>, <&gp 44>;
qcom,pin-func = <1>;
qcom,num-grp-pins = <5>;
label = “wcnss-5wire-active”;
wcnss-5wire-active: wcnss-active {
drive-strength = <6>; / * 6MA */
bias-pull-up;
};
```

```
};
Pmx_wcnss_5wire_suspend {
qcom,pins = <&gp 40>, <&gp 41>, <&gp 42>, <&gp 43>, <&gp 44>;
qcom,pin-func = <0>;
qcom,num-grp-pins = <5>;
label = “wcnss-5wire-suspend”;
wcnss-5wire-sleep: wcnss-sleep {
drive-strength = <6>; / * 6MA */
bias-pull-down;
};
};
```

5.1.4 Pin group configuration for SDC pins (runtime)

TLMM Ver 4 supports dedicated SDC pin configurations for SDC1 and SDC2. These pins are used for clock, command, and data signals as part of the SDC specification. QTI models this as 6 SDC type pins. Pins 0 to 2 correspond to clock, command, and data of SDC1; pins 3 to 5 correspond to clock, command, and data of SDC2. The following example describes the configuration of SDC pin types’ pins:

1. Extend the pinmux node to define pins for SDC1 and SDC2.

```
&tlmm_pinmux {
...
..
     /* SDC pin type */
     sdc: sdc {
          qcom,pin-type-sdc;
          /* 0-2 for sdc1 4-6 for sdc2 */
          qcom,num-pins = <7>;
          /* Order of pins */
          /* SDC1: CLK -> 0, CMD -> 1, DATA -> 2 */
          /* SDC2: CLK -> 4, CMD -> 5, DATA -> 6 */
          #qcom,pin-cells = <1>;
     };

     pmx_sdc1_clk {
          qcom,pins = <&sdc 0>;
          qcom,num-grp-pins = <1>;
          label = "sdc1-clk";
          sdc1_clk_on: clk_on {
               bias-disable; /* NO pull */
               drive-strength = <16>; /* 16 MA */
          };
          sdc1_clk_off: clk_off {
               bias-disable; /* NO pull */
```

```
            drive-strength = <2>; /* 2 MA */
        };
    };
    pmx_sdc1_cmd {
        qcom,pins = <&sdc 1>;
        qcom,num-grp-pins = <1>;
        label = "sdc1-cmd";
        sdc1_cmd_on: cmd_on {
            bias-pull-up; /* pull up */
            drive-strength = <10>; /* 10 MA */
        };
        sdc1_cmd_off: cmd_off {
            bias-pull-up; /* pull up */
            drive-strength = <2>; /* 2 MA */
        };
    };

    pmx_sdc1_data {
        qcom,pins = <&sdc 2>;
        qcom,num-grp-pins = <1>;
        label = "sdc1-data";
        sdc1_data_on: data_on {
            bias-pull-up; /* pull up */
            drive-strength = <10>; /* 10 MA */
        };
        sdc1_data_off: data_off {
            bias-pull-up; /* pull up */
            drive-strength = <2>; /* 2 MA */
        };
    };

    pmx_sdc2_clk {
        qcom,pins = <&sdc 4>;
        qcom,num-grp-pins = <1>;
        label = "sdc2-clk";
        sdc2_clk_on: clk_on {
            bias-disable; /* NO pull */
            drive-strength = <16>; /* 16 MA */
        };
        sdc2_clk_off: clk_off {
            bias-disable; /* NO pull */
            drive-strength = <2>; /* 2 MA */
        };
    };
```

```
pmx_sdc2_cmd {
     qcom,pins = <&sdc 5>;
     qcom,num-grp-pins = <1>;
     label = "sdc2-cmd";
     sdc2_cmd_on: cmd_on {
          bias-pull-up; /* pull up */
          drive-strength = <10>; /* 10 MA */
     };
     sdc2_cmd_off: cmd_off {
          bias-pull-up; /* pull up */
          drive-strength = <2>; /* 2 MA */
     };
};

pmx_sdc2_data {
     qcom,pins = <&sdc 6>;
     qcom,num-grp-pins = <1>;
     label = "sdc2-data";
     sdc2_data_on: data_on {
          bias-pull-up; /* pull up */
          drive-strength = <10>; /* 10 MA */
     };
     sdc2_data_off: data_off {
          bias-pull-up; /* pull up */
          drive-strength = <2>; /* 2 MA */
     };
};
```

2. Update the SDHC nodes to point to the pin configurations defined in step 1. Delete the old configurations.

```
&sdhc1 {
…..
…
qcom,pad-pull-on = <0x0 0x3 0x3>; /* no-pull, pull-up, pull-up */
qcom,pad-pull-off = <0x0 0x3 0x3>; /* no-pull, pull-up, pull-up */
qcom,pad-drv-on = <0x4 0x4 0x4>; /* 10mA, 10mA, 10mA */
qcom,pad-drv-off = <0x0 0x0 0x0>; /* 2mA, 2mA, 2mA */
pinctrl-names = “on”, “off”;
pinctrl-0 = <&sdc1_clk_on &sdc1_cmd_on &sdc1_data_on>;
pinctrl-1 = <&sdc1_clk_off, &sdc1_cmd_on &sdc1_data_on>;
};
```

```
&sdhc2 {
…..
…
qcom,pad-pull-on = <0x0 0x3 0x3>; /* no-pull, pull-up, pull-up */
qcom,pad-pull-off = <0x0 0x3 0x3>; /* no-pull, pull-up, pull-up */
qcom,pad-drv-on = <0x4 0x4 0x4>; /* 10mA, 10mA, 10mA */
qcom,pad-drv-off = <0x0 0x0 0x0>; /* 2mA, 2mA, 2mA */
pinctrl-names = “on”, “off”;
pinctrl-0 = <&sdc2_clk_on &sdc2_cmd_on &sdc2_data_on>;
pinctrl-1 = <&sdc2_clk_off, &sdc2_cmd_on &sdc2_data_on>;
};
```

3. Update the driver to parse device tree for pinctrl nodes as shown in step 3 of section 5.1.2.

5.2 Use case 2 – Pins driven by software (bit bang)

The following two frameworks apply to this use case:

- Pinctrl to configure the pin
- GPIO lib to drive the required signal values on the pin

The general-purpose pin type on the TLMM node (currently the only pin type on TLMM Ver 4 to support GPIO functionality) must be augmented with a GPIO chip controller node. Extend the TLMM pinmux node from chapter 4, as shown in the following example. The GPIO controller node is used as a handle to specify GPIO pins in client nodes. Such pins are typically used outside the SoC, e.g., display panel, Ethernet, etc. These pin group nodes are added by extending the tlmm_pinmux node in the corresponding <soc>-board.dtsi file.

The following example shows configuration of the TLMM pinmux node with the GPIO controller for the gp pin type:

```
&soc {
tlmm_pinmux: pinctrl@1000000 {
compatible = “qcom,msm-tlmm-v4”;
reg = <0x1000000 0x300000>;
/*General purpose pins*/
gp: gp {
qcom,pin-type-gp;
qcom,num-pins = <121>;
#qcom,pin-cells = <1>;
/* Add GPIO controller node */
msm_gpio: msm_gpio {
gpio-controller;
#gpio-cells = <2>;
};
};
```

```
/* SDC pin type */
sdc: sdc {
qcom,pin-type-sdc;
/* 0-2 for sdc1 3-5 for sdc2 */
qcom,num-pins = <6>;
/* Order of pins */
/* CLK -> 0 */
/* CMD -> 1 */
/* DATA -> 2 */
#qcom,pin-cells = <1>;
};
};
};
```

5.2.1 Configuring pins for software bit banging (runtime)

The following example shows GPIO client and pin configuration for Active and Sleep states:

1. Extend the tlmm_pinmux node in the corresponding <soc>-<board>.dtsi file to add pin configuration node for Active and Sleep states.

```
&tlmm_pinmux {
…..
….
Client3_pins {
/* Pin group node */
/* Uses general purpose pins */
qcom,pins = <&gp 0>
qcom,num-grp-pins = <1>;
/* function setting not required GPIO by default */
label = “client3-gpios”;
/* Active configuration of bus pins */
Client3_active: client3_active {
/* Configuration node:
• Property names as specified in
• pinctrl-bindings.txt /
drive-strength = <8>; /* 8 MA */
bias-pull-up; /* PULL UP */
};
Client3_suspend: client3_suspend {
/* Configuration node:
• Property names as specified in
• pinctrl-bindings.txt /
drive-strength = <2>; /* 2 MA */
bias-disable; /* No PULL */
```

```
};
};
```

2. Modify the client node to include pinctrl states. Also, modify any explicit references to the GPIO chip handler from section 5.2.

```
Soc {
…….
…
Client2 {
……
pinctrl-states = “active”, “sleep”;
/* Use phandle referrence to active and sleep configurations to
• Define the active and sleep pin states */
pinctrl-0 = <&Client3_active>;
pinctrl-1 = <&Client3_sleep>;
/* Use phandle of correct gpio controller */
client3_gpio_1 = <&msm_gpio 0 0>;
client3-gpio-1 = <&msmgpio 0 0>;
client3-gpio-2 = <&msmgpio 1 0>;
};
```

3. Modify the driver to look for pinctrl states only. The GPIOs are probed even after the pinctrl states are found

```
int client3_probe ( struct platform_device *pdev)
{
struct pinctrl_state *set_state;

…….
….
/* Try to obtain pinctrl handle */ 30
pdev->dev->pins->p = devm_pinctrl_get(pdev);
If (!pdev->dev->pins->p) {

/* Lookup the pin configuration */
set_state = pinctrl_lookup_state(pdev->dev->pins->p, “active”);
if (!set_state)
goto fail;
client3_data->active_state = set_state;
set_state = pinctrl_lookup_state(pdev->dev->pins->p, “suspend”);
client3_data->sleep_state = set_state;
```

```
(Optional) Actually write the active configuration to hardware*/
ret = pinctrl_select_state(pdev->dev->pins->p,
client3_data->active_state);
}
gpio_probe:
/* (Perform Regular probe of gpios. */
Client3_data->reset_gpio = of_get_named_gpio(node, “client3_gpio_1”);
Gpio_request(reset_gpio, “reset”);
…..
}
```

4. Install the required configuration before bit banging.

```
int Client3_drive_signal ( boolean active )
{
If (client3_data->use_pinctrl) {
If (active)
ret = pinctrl_select_state(pdev->dev->pins->p,
client3_data->active_state);
else
ret = pinctrl_select_state(pdev->dev->pins->p,
client3_data->sleep_state);
}
gpio_direction_out(client3_data->reset_gpio);
gpio_set_value(client3_data->reset_gpio, 1);
}
```

5.3 Use case 3 – Pins used as interrupt sources

The general-purpose pins present on a TLMM block can be used as interrupt sources. The pinmux node must be augmented with interrupt controller attributes to support this use case. Since this is supported by general-purpose pin types, augment the corresponding GPIO controller node with interrupt controller attributes.

The following example shows the pinmux node with interrupt controller attributes:

```
&soc {
tlmm_pinmux: pinctrl@1000000 {
compatible = “qcom,msm-tlmm-v4”;
reg = <0x1000000 0x300000>;
/*General purpose pins*/
gp: gp {
qcom,pin-type-gp;
qcom,num-pins = <121>;
```

```
#qcom,pin-cells = <1>;
/* Add GPIO controller node */
msm_gpio: msm_gpio {
gpio-controller;
#gpio-cells = <2>;
/* Add interrupt controller attributes */
compatible = “qcom,msm-tlmm-gp-intc”;
interrupt-controller;
#interrupt-cells = <2>;
num_irqs = <121>;
};
};
/* SDC pin type */
sdc: sdc {
qcom,pin-type-sdc;
/* 0-2 for sdc1 3-5 for sdc2 */
qcom,num-pins = <6>;
/* Order of pins */
/* CLK -> 0 */
/* CMD -> 1 */
/* DATA -> 2 */
#qcom,pin-cells = <1>;
};
};
};
```


5.3.1 Configuring pins to be used as interrupt sources only once (at bootup)

When pins are used as interrupt sources, the pin attributes are configured just once, at bootup. The pin is then used as an interrupt source. To accomplish this, the pin configuration must be modeled as a default state of the pin. Also, since these devices are present as attachments to the SoC on a board, their pin groups are added by extending the pinmux node in the corresponding board-specific device tree source file.

The following example shows an IRQ client:

1. Extend the tlmm_pinmux node in the <soc>-<board>.dtsi file to add the pin configuration.

```
&tlmm_pinmux {
…..
….
Client4_pins {
/* Pin group node */
/* Uses general purpose pins */
qcom,pins = <&gp 0>;
```

```
qcom,num-grp-pins = <1>;
/* function setting not required GPIO by default */
label = “client4-irq”;
/* Active configuration of bus pins */
Client4_default: client4_default {
/* Configuration node:
• Property names as specified in
• pinctrl-bindings.txt /
drive-strength = <8>; /* 8 MA */
bias-pull-up; /* PULL UP */
};
};
```

2. Update the client device node to point to the pin configuration node.

```
Soc {
…….
…
Client4 {
……
pinctrl-states = “default”;
/* Use phandle referrence to default config node above */
pinctrl-0 = <&client4_default>;
/* Remove existing reference to interrupt parent phandle */
interrupt-parent = <&msmgpio 0 2>;
/* Add reference to the interrupt controller node defined in example
above
interrupt-parent = <&msm_gpio 0 2>;
};
```

The client driver can proceed to request the interrupt as currently implemented.

Driver changes are not necessary since a default pin configuration is installed by the kernel at bootup.

6 GPIO debug using adb commands

NOTE: This chapter was added to this document revision.

Through Sysfs interface for userspace, you can control the GPIO direction and value. The control interfaces are write-only:

/sys/class/gpio/

- “export” – Userspace may ask the kernel to export control of a GPIO to userspace by writing its number to this file.
  - Example – "echo 19 > export" will create a "gpio19" node for GPIO #19, if that is not requested by kernel code.
- "unexport" – Reverses the effect of exporting to userspace.
  - Example – "echo 19 > unexport" will remove a "gpio19" node exported using the "export" file.

Once the GPIO is exported, the GPIO signals have paths like /sys/class/gpio/gpio19/ (for GPIO #19) and have the following read/write attributes:

/sys/class/gpio/gpioN/direction

- "direction" – Reads as either "in" or "out". This value may normally be written. Writing as "out" defaults to initializing the value as low.

/sys/class/gpio/gpioN/value

- "value" – reads as either 0 (low) or 1 (high). If the GPIO is configured as an output, this value may be written; any nonzero value is treated as high.

You can also read the GPIO config register through `adb cmd` and modify the configurations.

/system/bin/r <Address> <value>

```
TLMM_GPIO_CFGn    0x1000000 + 0x1000 * (n)
TLMM_GPIO_IN_OUTn      0x1000004 + 0x1000 * (n)
TLMM_GPIO_INTR_CFGn    0x1000008 + 0x1000 * (n)
TLMM_GPIO_INTR_STATUSn     0x100000C + 0x1000 * (n)
```

Example:

- To read GPIO 19 config registers

```
#/system/bin/r 0x1013000
#/system/bin/r 0x1013004
#/system/bin/r 0x1013008
#/system/bin/r 0x101300C
```

- To configure GPIO 19 as output high

```
#/system/bin/r 0x1013000 0x201
#/system/bin/r 0x1013004 0x3
```

A References

A.1 Related documents

Title	Number
Qualcomm Technologies, Inc.	
Resources	
PINCTRL (Pin Control) Subsystem	https://www.kernel.org/doc/Documentation/pinctrl.txt

A.2 Acronyms and terms

Acronym or term	Definition
GPIO	General Purpose Input Output
TLMM	Top-Level Mode Multiplexer