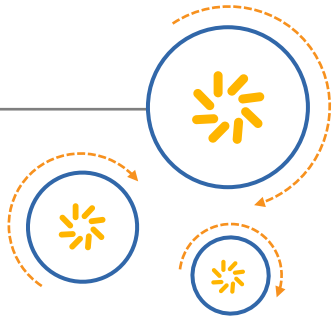
Qualcomm Technologies, Inc.



Linux BAM Low-Speed Peripherals Configuration and Debug Guide

80-NU767-1 H

July 15, 2016

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

Revision history

Revision	Date	Description
A	January 2015	Initial release
B	April 2015	Updated Chapters 3, 4, and 5
C	May 2015	Updated Sections 3.4, 5.2 and 7.4 Numerous other minor edits made throughout the document
D	August 2015	Updated Chapters 3, 4, 5, and 7
E	September 2015	Updated Tables 4-3 and 5-1
F	September 2015	Updated Section 3.4.1 and Tables 3-4, 4-3, 5-1 and 5-3
G	March 2016	Added Chapter 9; updated Table 5-5
H	July 2016	Added Sections 5.2.3 and 5.4 Added Tables 3-7, 4-7, and 5-7 Updated Sections 3.4.1 and 9.4 Updated Tables 3-4, 4-4, and 5-4

Contents

Figures

Tables

1 Introduction

1.1 Purpose

This document describes how to configure, use, and debug the Bus Access Manager (BAM) Low-Speed Peripherals (BLSP) on the Linux platform available on the MSM8996, MSM8994, MSM8992, MSM8952, MSM8916, MSM8936/ MSM8939, MSM8909, MDM9x35, and MDM9x40/MDM9x45 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>`.

Commands to be entered appear in a different font, for example, **`copy a:*.* b:`**.

Button and key names appear in bold font, for example, click **Save** or press **Enter**.

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 Device tree

Device tree is a standard used by Open Firmware to represent hardware. Instead of compiling multiple board support package files into the kernel, a separate OS-independent binary describes the target. The data structure is loaded into the operating system at boot time. Device tree is composed of trees, nodes, and properties that are similar to XML.

The advantages and disadvantages of the device tree are summarized below.

Pros	Cons
▪ Formal and clear hardware description ▪ Multiplatform kernels are possible ▪ Less board-specific code, more efficient device driver binding	▪ Not a complete built-in dependency solution

For more detailed information on device tree, refer to the Device Tree Wiki (http://www.devicetree.org/Main_Page{Device Tree Wiki).

2.1 Device tree components

Table 2-1 Device tree components

Component	Description
Source (*.dts)	Expresses the device tree in human-editable format; it is organized as a tree structure of nodes and properties. For ARM architecture, the source is in the dts folder: `kernel/arch/arm/boot/dts` Files with the `.dtsi` extension are device tree include files. They are useful for factoring out details that do not change between boards or hardware revisions.
Bindings	Defines how a device is described in the device tree; see the bindings folder for documentation: `kernel/Documentation/devicetree/bindings`
Device Tree Blob (*.dtb)	Compiled version of the device source; it is also known as the Flattened Device Tree. The DTS Compiler Tool (DTC) compiles DTS to DTB. Starting with the MSM8974 and later chipsets, QTI only supports the device tree in our drivers to be compliant with upstream Linux submission.

Chip-specific components	Chipset-specific files include the chip ID as shown in the following examples: ▪ Main DTS that contains chipset and peripheral information that is common for all hardware variants (CDP, MTP, etc.): ▫ kernel/arch/arm/boot/dts/qcom/msm8974.dtsi ▫ kernel/arch/arm/boot/dts/qcom/msm8226.dtsi ▫ kernel/arch/arm/boot/dts/qcom/apq8084.dtsi ▫ kernel/arch/arm/boot/dts/msm9625.dtsi ▪ DTS file that is used by the CDP: ▫ kernel/arch/arm/boot/dts/qcom/msm8974-cdp.dts ▫ kernel/arch/arm/boot/dts/qcom/msm8226-v1-cdp.dtsi ▫ kernel/arch/arm/boot/dts/qcom/apq8084-cdp.dtsi ▫ kernel/arch/arm/boot/dts/msm9625-cdp.dtsi ▪ BUS Scale Topology (ID) list: ▫ kernel/arch/arm/boot/dts/qcom/msm8974-bus.dtsi ▫ kernel/arch/arm/boot/dts/qcom/msm8226-bus.dtsi ▫ kernel/arch/arm/boot/dts/qcom/apq8084-bus.dtsi

3 UART

This chapter describes the UART and explains how to configure it in the boot loader and kernel.

3.1 Hardware overview

BLSP

BLSP is a new design that replaces the legacy BLSP core in the new chipset families. All legacy BLSP software registers have been removed.

Each BLSP block includes six QUP and six UART cores. Instead of the legacy data mover (ADM), BAM is used as a hardware data mover. Each BLSP peripheral:

- Is statically connected to a pair of BAM pipes
- Consists of 24 pipes that can be used for data move operations
- Supports BAM and non-BAM based data transfers

UART core

Key features added for the chipset:

- BAM support
- Single-character mode
- Baudrates 300 bps up to 4M bps
  - Detail information in msm_hsl_set_baud_rate() of kernel/drivers/tty/serial/msm_serial_hs_lite.c
  - Detail information in msm_hs_set_bps_locked() of kernel/drivers/tty/serial/msm_serial_hs.c

For performance purpose, QTI highly recommends to use high speed UART driver for high baudrate and/or big data package; for example, Bluetooth connecitivity.

UART core is used to transmit and receive data through a serial interface. It is used to communicate with other UART protocol devices. Configuration of this mode is primarily defined by the UART_DM_MR1 and UART_DM_MR2 registers.

To match the labeling in the software interface manual, each UART is identified by the BLSP core (1 or 2) and UART core (0 to 5). The max transfer rate of the UART core is up to 4M bps.

Table 3-1 UART_DM physical address, IRQ numbers, Kernel UART clock name, consumer, producer pipes, BLSP_BAM physical address, and BAM IRQ number for MSM8994 and MSM8992

BLSP hardware ID	UART_DM core	Physical address (UART_DM_BASE_ADDRESS), size	IRQ #	Bus master ID	Kernel UART clock name	Consumer, producer pipes	BLSP_BAM physical address, size, IRQ number
BLSP 1	BLSP 1 UART 0	0xF991D000,0x1000	107	86	clock_gcc_blsp1_uart1_apps_clk	0, 1	0xF9904000,0x19000,238
BLSP 2	BLSP 1 UART 1	0xF991E000,0x1000	108	86	clock_gcc_blsp1_uart2_apps_clk	2, 3	0xF9904000, 0x19000,238
BLSP 3	BLSP 1 UART 2	0xF991F000,0x1000	109	86	clock_gcc_blsp1_uart3_apps_clk	4, 5	0xF9904000, 0x19000,238
BLSP 4	BLSP 1 UART 3	0xF9920000,0x1000	110	86	clock_gcc_blsp1_uart4_apps_clk	6, 7	0xF9904000, 0x19000,238
BLSP 5	BLSP 1 UART 4	0xF9921000,0x1000	111	86	clock_gcc_blsp1_uart5_apps_clk	8, 9	0xF9904000, 0x19000,238
BLSP 6	BLSP 1 UART 5	0xF9922000,0x1000	112	86	clock_gcc_blsp1_uart6_apps_clk	10, 11	0xF9904000, 0x19000,238
BLSP 7	BLSP 2 UART 0	0xF995D000,0x1000	113	84	clock_gcc_blsp2_uart1_apps_clk	0, 1	0xF9944000, 0x19000,239
BLSP 8	BLSP 2 UART 1	0xF995E000,0x1000	114	84	clock_gcc_blsp2_uart2_apps_clk	2, 3	0xF9944000, 0x19000,239
BLSP 9	BLSP 2 UART 2	0xF995F000,0x1000	115	84	clock_gcc_blsp2_uart3_apps_clk	4, 5	0xF9944000, 0x19000,239
BLSP10	BLSP 2 UART 3	0xF9960000,0x1000	116	84	clock_gcc_blsp2_uart4_apps_clk	6, 7	0xF9944000, 0x19000,239
BLSP11	BLSP 2 UART 4	0xF9961000,0x1000	117	84	clock_gcc_blsp2_uart5_apps_clk	8, 9	0xF9944000, 0x19000,239
BLSP12	BLSP 2 UART 5	0xF9962000,0x1000	118	84	clock_gcc_blsp2_uart6_apps_clk	10, 11	0xF9944000, 0x19000,239

Table 3-2 UART_DM physical address, IRQ numbers, Kernel UART clock name, consumer, producer pipes, BLSP_BAM physical address, and BAM IRQ number for MSM8916, MSM8936, MSM8939, MSM8909

BLSP hardware ID	UART_DM core	Physical address (UART_DM_BASE_ADDRESS), size	IRQ#	Bus master ID	Kernel UART clock name	Consumer, producer pipes	BLSP_BAM physical address, size, IRQ number
BLSP 1	BLSP 1 UART 0	0x78AF000,0x200	107	86	clock_gcc_blsp1_uart1_apps_clk	0, 1	0x7884000, 0x23000, 238
BLSP 2	BLSP 1 UART 1	0x78B0000,0x200	108	86	clock_gcc_blsp1_uart2_apps_clk	2, 3	0x7884000, 0x23000, 238

Table 3-3 UART_DM physical address, IRQ numbers, Kernel UART clock name, consumer, producer pipes, BLSP_BAM physical address, and BAM IRQ number for MSM8996

BLSP hardware ID	UART_DM core	Physical address (UART_DM_BASE_ADDRESS), size	IRQ#	Bus master ID	Kernel UART clock name	Consumer, producer pipes	BLSP_BAM physical address, size, IRQ number
BLSP 1	BLSP 1 UART 0	0x756F000,0x1000	107	86	clock_gcc_blsp1_uart1_apps_clk	0, 1	0x7544000, 0x2B000, 238
BLSP 2	BLSP 1 UART 1	0x7570000,0x1000	108	86	clock_gcc_blsp1_uart2_apps_clk	2, 3	0x7544000, 0x2B00, 238
BLSP 3	BLSP 1 UART 2	0x7571000,0x1000	109	86	clock_gcc_blsp1_uart3_apps_clk	4, 5	0x7544000, 0x2B00, 238
BLSP 4	BLSP 1 UART 3	0x7572000,0x1000	110	86	clock_gcc_blsp1_uart4_apps_clk	6, 7	0x7544000, 0x2B00, 238
BLSP 5	BLSP 1 UART 4	0x7573000,0x1000	111	86	clock_gcc_blsp1_uart5_apps_clk	8, 9	0x7544000, 0x2B00, 238
BLSP 6	BLSP 1 UART 5	0x7574000,0x1000	112	86	clock_gcc_blsp1_uart6_apps_clk	10, 11	0x7544000, 0x2B00, 238
BLSP 7	BLSP 2 UART 0	0x75AF000,0x1000	113	84	clock_gcc_blsp2_uart1_apps_clk	0, 1	0x7584000, 0x2B00, 239
BLSP 8	BLSP 2 UART 1	0x75B0000,0x1000	114	84	clock_gcc_blsp2_uart2_apps_clk	2, 3	0x7584000, 0x2B00, 239
BLSP 9	BLSP 2 UART 2	0x75B1000,0x1000	115	84	clock_gcc_blsp2_uart3_apps_clk	4, 5	0x7584000, 0x2B00, 239
BLSP10	BLSP 2 UART 3	0x75B2000,0x1000	116	84	clock_gcc_blsp2_uart4_apps_clk	6, 7	0x7584000, 0x2B00, 239
BLSP11	BLSP 2 UART 4	0x75B3000,0x1000	117	84	clock_gcc_blsp2_uart5_apps_clk	8, 9	0x7584000, 0x2B00, 239
BLSP12	BLSP 2 UART 5	0x75B4000,0x1000	118	84	clock_gcc_blsp2_uart6_apps_clk	10, 11	0x7584000, 0x2B00, 239

Table 3-4 UART_DM physical address, IRQ numbers, Kernel UART clock name, consumer, producer pipes, BLSP_BAM physical address, and BAM IRQ number for MSM8952, MSM8953, and MSM8956/76

NOTE: The following table has been updated.

BLSP hardware ID	UART_DM core	Physical address (UART_DM_BASE_ADDRESS), size	IRQ#	Bus master ID	Kernel UART clock name	Consumer, producer pipes	BLSP_BAM physical address, size, IRQ number
BLSP 1	BLSP 1 UART 0	0x78AF000,0x1000	107	86	clock_gcc_blsp1_uart1_apps_clk	0, 1	0x7884000, 0x23000, 238
BLSP 2	BLSP 1 UART 1	0x78B0000,0x1000	108	86	clock_gcc_blsp1_uart2_apps_clk	2, 3	0x7884000. 0x23000. 238
BLSP 3	BLSP 2 UART 0	0x7AEF000,0x1000	306	84	clock_gcc_blsp2_uart1_apps_clk	0, 1	0x7AC4000. 0x23000, 239
BLSP 4	BLSP 2 UART 1	0x7AF0000,0x1000	307	84	clock_gcc_blsp2_uart2_apps_clk	2, 3	0x7AC4000. 0x23000, 239

Table 3-5 UART_DM physical address, IRQ numbers, Kernel UART clock name, consumer, producer pipes, BLSP_BAM physical address, and BAM IRQ number for MDM9x35

BLSP hardware ID	UART_DM core	Physical address (UART_DM_BASE_ ADDRESS), size	IRQ#	Bus master ID	Kernel UART clock name	Consumer, producer pipes	BLSP_BAM physical address, size, IRQ number
BLSP 1	BLSP 1 UART 0	0xF991D000,0x1000	107	86	clock_gcc_blsp1_uart1_apps_clk	0, 1	0xF9904000, 0x19000, 238
BLSP 2	BLSP 1 UART 1	0xF991E000,0x1000	108	86	clock_gcc_blsp1_uart2_apps_clk	2, 3	0xF9904000, 0x19000, 238
BLSP 3	BLSP 1 UART 2	0xF991F000,0x1000	109	86	clock_gcc_blsp1_uart3_apps_clk	4, 5	0xF9904000, 0x19000, 238
BLSP 4	BLSP 1 UART 3	0xF9920000,0x1000	110	86	clock_gcc_blsp1_uart4_apps_clk	6, 7	0xF9904000, 0x19000, 238
BLSP 5	BLSP 1 UART 4	0xF9921000,0x1000	111	86	clock_gcc_blsp1_uart5_apps_clk	8, 9	0xF9904000, 0x19000, 238
BLSP 6	BLSP 1 UART 5	0xF9922000,0x1000	112	86	clock_gcc_blsp1_uart6_apps_clk	10, 11	0xF9904000, 0x19000, 238

Table 3-6 UART_DM physical address, IRQ numbers, Kernel UART clock name, consumer, producer pipes, BLSP_BAM physical address, and BAM IRQ number for MDM9x40/9x45, and MDM9x50/9x55

BLSP hardware ID	UART_DM core	Physical address (UART_DM_BASE_ ADDRESS), size	IRQ#	Bus master ID	Kernel UART clock name	Consumer, producer pipes	BLSP_BAM physical address, size, IRQ number
BLSP 1	BLSP 1 UART 0	0x78AF000,0x200	107	86	clock_gcc_blsp1_uart1_apps_clk	0, 1	0x7884000, 0x23000, 238
BLSP 2	BLSP 1 UART 1	0x78B0000,0x200	108	86	clock_gcc_blsp1_uart2_apps_clk	2, 3	0x7884000, 0x23000, 238
BLSP 3	BLSP 1 UART 2	0x78B1000,0x200	109	86	clock_gcc_blsp1_uart3_apps_clk	4, 5	0x7884000, 0x23000, 238
BLSP 4	BLSP 1 UART 3	0x78B2000,0x200	110	86	clock_gcc_blsp1_uart4_apps_clk	6, 7	0x7884000, 0x23000, 238

Table 3-7 UART_DM physical address, IRQ numbers, Kernel UART clock name, consumer, producer pipes, BLSP_BAM physical address, and BAM IRQ number for MDM9x07

NOTE: The following table was added to this document revision.

BLSP hardware ID	UART_DM core	Physical address (UART_DM_BASE_ ADDRESS), size	IRQ#	Bus master ID	Kernel UART clock name	Consumer, producer pipes	BLSP_BAM physical address, size, IRQ number
BLSP 1	BLSP 1 UART 0	0x78AF000,0x200	107	86	clock_gcc_blsp1_uart1_apps_clk	0, 1	0x7884000, 0x23000, 238
BLSP 2	BLSP 1 UART 1	0x78B0000,0x200	108	86	clock_gcc_blsp1_uart2_apps_clk	2, 3	0x7884000, 0x23000, 238
BLSP 3	BLSP 1 UART 2	0x78B1000,0x200	119	86	clock_gcc_blsp1_uart3_apps_clk	4, 5	0x7884000, 0x23000, 238
BLSP 4	BLSP 1 UART 3	0x78B2000,0x200	120	86	clock_gcc_blsp1_uart4_apps_clk	6, 7	0x7884000, 0x23000, 238
BLSP 5	BLSP 1 UART 4	0x78B3000,0x200	121	86	clock_gcc_blsp1_uart5_apps_clk	8, 9	0x7884000, 0x23000, 238
BLSP 6	BLSP 1 UART 5	0x78B4000,0x200	122	86	clock_gcc_blsp1_uart6_apps_clk	10, 11	0x7884000, 0x23000, 238

3.2 Configure LK UART

In the Little Kernel (LK) boot loader, a UART may be needed for debug logs.

3.2.1 Code changes

This section describes the changes required to configure a UART in the LK boot loader. The following files are used to configure UART in the boot loader:

```
/bootable/bootloader/lk/project/<chipset>.mk
/bootable/bootloader/lk/target/<chipset>/init.c
/bootable/bootloader/lk/platform/<chipset>/include/platform/iomap.h
/bootable/bootloader/lk/platform/<chipset>/acpuclock.c
/bootable/bootloader/lk/platform/<chipset>/<chipset>-clock.c
/bootable/bootloader/lk/platform/<chipset>/gpio.c
/kernel/arch/arm/mach-msm/include/mach/msm_iomap-<chip>.h
```

Where <*chipset*> corresponds to the applicable chipset, and <*chip*> corresponds to the 4-digit chip number, for example:

```
/bootable/bootloader/lk/project/msm8909.mk
/kernel/arch/arm/mach-msm/include/mach/msm_iomap-8909.h
```

1. Enable the UART for debugging.
   a. Open the project make file.

      `Project_Root/bootable/bootloader/lk/project/<chipset>.mk`

      Where <*chipset*> corresponds to the applicable chipset, for example:

      `Project_Root/bootable/bootloader/lk/project/msms8909.mk`

   b. Set the `WITH_DEBUG_UART` flag to TRUE.

      ```
      DEFINES += WITH_DEBUG_UART=1
      ```

2. Set the base address.
   a. Open the init.c file located at:

      `Project_Root/bootable/bootloader/lk/target/<chipset>/init.c`

      Where <*chipset*> corresponds to the applicable chipset, for example:

```
Project_Root/bootable/bootloader/lk/target/msms8909/init.c
```

b. Set the applicable parameters for the base address. The following example illustrates setting the base address.

```
  void target_early_init(void)
  {
  #if WITH_DEBUG_UART
          uart_dm_init(2, 0, BLSP1_UART1_BASE);
  #endif
  }
       void uart_dm_init(uint8_t id, uint32_t gsbi_base, uint32_t
uart_dm_base)
   id:  Represents BLSP ID( 1- 12).
   gsbi_base:  Set it to 0 if platform has BLSP instead of GSBI.
       uart_dm_base: Physical address for UART CORE defined in
/bootable/bootloader/lk/platform/(platform_name)/include/platform/iom
ap.h
```

3. Configure the clocks. Modify the `acpuclock.c` file located at:

```
Project_Root/bootable/bootloader/lk/platform/<chipset>/acpuclock.c
```

Where *<chipset>* corresponds to the applicable chipset, for example:

```
Project_Root/bootable/bootloader/lk/platform/MSM8909/acpuclock.c
```

The following example illustrates enabling the BLSP AHB and UART core clocks. These clocks are both required for UART to function correctly on the MSM8909 device.

```
/*
NOTE: Implementation of this function might be slightly different between
      different chipsets.
*/
void clock_config_uart_dm(uint8_t id)
{
  int ret;
  /*
    NOTE: In clock regime clocks are # from 1 to 6 so UART0 would
    be identified as UART1
  */
  //iface_clk is BLSP clk
  ret = clk_get_set_enable("uart2_iface_clk", 0, 1);

  //core_clock is UART clock.
  ret = clk_get_set_enable("uart2_core_clk", 7372800, 1);
}
```

4. Register the clocks with the clock regime. The BLSP1_AHB clock is enabled by default. To use the BLSP2_AHB clock, make the following modifications.

   a. Add the physical addresses to the `iomap.h` file located at:

      `Project_Root/bootable/bootloader/lk/platform/msm8994/include/platform/iomap.h`

      The following example illustrates addition of the BLSP2_AHB clock.

```
#define BLSP2 AHB CBCR          (CLK CTL BASE + 0x944) //0xFC400944
```

   b. Open the `<chipset>-clock.c` file located at:

      `Project_Root/bootable/bootloader/lk/platform/<chipset>/<chipset>-clock.c`

      Where `<chipset>` corresponds to the applicable chipset, for example:

      `Project_Root/bootable/bootloader/lk/platform/msm8994/msm8994-clock.c`

   c. Create a new clock entry.

```
//Project_Root/bootable/bootloader/lk/platform/msm8994/msm8994-clock.c
//Use gcc_blsp1_ahb_clk as an example and define gcc_blsp1_ahb_clk
static struct local_vote_clk gcc_blsp1_ahb_clk = {
  .cbcr_reg     = (uint32_t *) BLSP2_AHB_CBCR,
  .vote_reg     = APCS_CLOCK_BRANCH_ENA_VOTE,
  .en_mask      = BIT(15), //Bit 15 is for BLSP2_AHB clk

  .c = {
    .dbg_name = "gcc_blsp2_ahb_clk",
    .ops      = &clk_ops_vote,
  },
};
```

   d. Register uart_iface clk (BLSP_AHB clk) with the clock driver by adding it to the clock table.

```
//Project_Root/bootable/bootloader/lk/platform/msm8994/msm8994-clock.c
static struct clk_lookup msm_clocks_8994[] =
{
  //Name should be same as one you add on clock_config_uart_dm
  CLK_LOOKUP("uart#_iface_clk", gcc_blsp2_ahb_clk.c), //New entry
```

e. Register the uart_core clock with the clock driver by adding it to the clock table.

```
//Project_Root/bootable/bootloader/lk/platform/msm8994/msm8994-clock.c
static struct clk_lookup msm_clocks_8994[] =
{
  ...
  //Name should be same as one you add on clock_config_uart_dm
  CLK LOOKUP("uart1 core clk",  gcc blsp1 uart1 apps clk.c),
```

By default, UART0 to UART2 are predefined on BLSP1 to be used by the boot loader. However, any of the 12 UART cores can be configured to be used by the boot loader. To use other UART cores, register the UART core clock with the clock regime. Contact QTI to obtain register definitions for the rest of the cores.

5. Configure the GPIO.

   a. Please refer to the chipset device specification to check BLSP gpio assignment, and refer to the chipset software interface to check BLSP gpio configuration.

   b. Open the gpio.c file located at:

```
Project_Root/bootable/bootloader/lk/platform/<chipset>/gpio.c
```

   c. Configure the correct GPIO.

```
void gpio_config_uart_dm(uint8_t id)
{
  /*
  Configure the RX/TX GPIO
  Argument 1: GPIO #
  Argument 2: Function (Please see device pinout for more information)
  Argument 3: Input/Output (Can be 0/1)
  Argument 4: Should be no PULL
  Argument 5: Drive strength
  Argument 6: Output Enable (Can be 0/1)
  */
  gpio_tlmm_config(5, 2, GPIO_INPUT, GPIO_NO_PULL,
  GPIO_8MA, GPIO_DISABLE);
  gpio_tlmm_config(4, 2, GPIO_OUTPUT, GPIO_NO_PULL,
  GPIO_8MA, GPIO_DISABLE);
}
```

NOTE: See the device pinout for information about the GPIO function. BLSPs 4, 5, 6, 7, 9, and 11 have different function assignments compared to other BLSPs.

6. Configure Early Printk – Not applicable to LE targets.

   Additional changes are needed during kernel configuration if the following features are enabled in the `/kernel/arch/arm/configs/<chipset>_defconfig` file:

   - CONFIG_DEBUG_LL=y
   - CONFIG_EARLY_PRINTK=y

There is a dependency between UART configuration on the little kernel and the Early Printk driver in the kernel. If the configuration settings listed above are enabled, the following message is displayed using the Early Printk driver:

```
"Uncompressing Linux..."
```

The message output is defined in the Early Printk driver.

```
void
decompress_kernel(unsigned long output_start, unsigned long free_mem_ptr_p,
            unsigned long free_mem_ptr_end_p,
            int arch_id)
{
      int ret;

        ...
      arch_decomp_setup();

      putstr("Uncompressing Linux..."); //uses early printk driver
      ret = do_decompress(input_data, input_data_end - input_data,
        ...
}
```

a. The Early Printk driver depends on the little kernel to configure the UART port. Open the `msm_iomap-8994.h` file located at:

   `Project_Root/kernel/arch/arm/mach-msm/include/mach/msm_iomap-<chip>.h`

   Where `<chip>` corresponds to the 4-digit chip number, for example:

   `Project_Root/kernel/arch/arm/mach-msm/include/mach/msm_iomap-8994.h`

b. Ensure that the UART port being configured in the little kernel is the same UART port that is used by the kernel.

```
#ifdef CONFIG_DEBUG_MSM8994_UART
#define MSM_DEBUG_UART_BASE     IOMEM(0xFA71E000) /* Keep this same*/
#define MSM_DEBUG_UART_PHYS     0xF991E000 /* Change to same UART base
address that's configured in LK*/
#endif
```

3.2.2 Debug LK UART

If the UART is properly configured, the following message appears on the serial console:

```
Android Bootloader - UART_DM Initialized!!!
```

If you do not see the message, use the debugging described in this section.

1. Set a breakpoint. In the `/bootable/bootloader/lk/platform/msm_shared/uart_dm.c` file, set a breakpoint inside the `uart_dm_init` function line at `msm_boot_uart_dm_write(uart_dm_base, data, 44)`.
2. Check the clock status. Run the `testclock.cmm` file to ensure that the BLSP_AHB clock, Peripheral Network on a Chip (PNoC) clock, and UART core clock are on. The `testclock.cmm` script dumps all clock settings. The script is located at:

   `rpm_proc/core/systemdrivers/clock/scripts/<chipset>/testclock.cmm`

   Where <*chipset*> corresponds to the applicable product, for example:
   - `rpm_proc/core/systemdrivers/clock/scripts/msm8994/testclock.cmm`
   - `rpm_proc/core/systemdrivers/clock/scripts/msm8909/testclock.cmm`

   Clocks are named as follows:

UART core clocks	gcc_blsp1:2_uart1:6_apps_clk blsp 1 or 2 uart 1 to 6 ex: gcc_blsp1_uart1_apps_clk
PNoC clock	gcc_periph_noc_ahb_clk
BLSP AHB clock	gcc_blsp1:2_ahb_clk blsp 1 or 2 ex: gcc_blsp1_ahb_clk

3. Verify that the GPIOs are correctly configured. Check the GPIO configuration register, GPIO_CFGn, to ensure that the GPIO settings are valid.

```
Physical Address: 0xFD511000 + (0x10 * n) = GPIO_CFGn
  n = GPIO #
  Example Address:
    0xFD511000 = GPIO_CFG0
    0xFD511010 = GPIO_CFG1

Bit definition for GPIO_CFGn
  Bits 31:11  Reserved
  Bit  10     GPIO_HIHYS_EN    Control the hihys_EN for GPIO
  Bit  9      GPIO_OE          Controls the Output Enable for GPIO
                               when in GPIO mode.
```

```
Bits 8:6    DRV_STRENGTH     Control Drive Strength
                             000:2mA  001:4mA  010:6mA  011:8mA
                             100:10mA 101:12mA 110:14mA 111:16mA
Bits 5:2    FUNC_SEL         Make sure Function is BLSP
                             Check Device Pinout for Correct Function
Bits 1:0    GPIO_PULL        Internal Pull Configuration
                             00:No Pull   01: Pull Down
                             10:Keeper    11: Pull Up
```

NOTE: For UART, 8 mA with no pull is recommended.

3.3 Configure kernel low-speed UART

Low-speed UART driver (kernel/drivers/tty/serial/msm_serial_hs_lite.c) is a FIFO-based UART driver and is designed to support small data transfer at a slow rate, such as for console debugging or IrDA transfer. The high-speed UART driver (kernel/drivers/tty/serial/msm_serial_hs.c) is a BAM-based driver and should be used if a large amount of data is transferred or for situations where a high-speed transfer is required.

3.3.1 Code changes

The following files are used to configure BLSP1 UART1 to use the low-speed UART driver:

File type	Description
Device tree source	For MSM™ and APQ products: /kernel/arch/arm/boot/dts/qcom/<*chipset*>.dtsi Where <*chipset*> corresponds to the applicable chipset, for example: /kernel/arch/arm/boot/dts/qcom/msm8909.dtsi
Clock table	The clock nodes need to be added to the DTSI file. For reference, the clocks are defined in: /kernel/drivers/clk/qcom/clock-gcc-<chipset>.c For example: /kernel/drivers/clk/qcom/clock-gcc-8994.c
Pinctrl settings	The pin control table is located in the following file: /kernel/arch/arm/boot/dts/qcom/<*chipset*>-pinctrl.dtsi

The following procedure describes how to configure BLSP1 UART1 to use the low-speed UART driver using the MSM8994 chipset as an example.

1. Create a device tree node.
   a. Please refer to the chipset device specification to check BLSP gpio assignment, and refer to the chipset software interface to check BLSP gpio configuration.
   b. Open the <*chipset*>.dtsi file located at:

      /kernel/arch/arm/boot/dts/qcom/<*chipset*>.dtsi

      Where <*chipset*> corresponds to the applicable chipset, for example:

```
/kernel/arch/arm/boot/dts/qcom/msm8994.dtsi
```

c. Add a new device tree node as illustrated in the following example with MSM8994 platform.

```
/* If multiple UARTs are registered, add aliases to identify the UART ID.*/
aliases {
  serial2 = &uart2; //uart2 will be registered as ttyHSL2
};


uart2:serial@f991e000 { //0xF991E000 is the UART_DM Base Address
  compatible = "qcom,msm-lsuart-v14"; //manufacture, model of serial driver
  reg = <0xf991e000 0x1000>; //Base address UART_DM and size
  /* First Field: 0 SPI interrupt (Shared Peripheral Interrupt)
     Second Field: Interrupt #
     Third field: Trigger type, keep 0
     See also:/kernel/Documentation/devicetree/bindings/arm/gic.txt   */
  interrupts = <0 108 0>;
  status = "ok"; //Status OK enables it
  /* Documentation/devicetree/bindings/arm/msm/msm_bus.txt
     name = Client name Keep same name as interface
     num-cases = # of use-cases, keep as 2.
     active-only = false (support dual context active sleep)
     num-paths = 1 for Master-Slave Pairs.
     vectors-KBps = master-id, slave-id, bandwidth settings..
     master-id = 86 for BLSP1
                 84 for BLSP2
     EBI CH0 slave-id = 512
     Get master ID from /kernel/arch/arm/boot/dts/qcom/msm8994.dtsi
     bandwidth settings: Keep @ below settings "0 0", "500 800"    */
  qcom,msm-bus,name = "serial_uart2";
  qcom,msm-bus,num-cases = <2>;
  qcom,msm-bus,num-paths = <1>;
  qcom,msm-bus,vectors-KBps =
       <86 512 0 0>,
       <86 512 500 800>;
  /* Select nodes for AHB and Core clocks
     In clock regime UART clocks are labeled #1 to #6. So BLSP1 UART1 is
     clk_gcc_blsp1_uart2_apps_clk   */
  clock-names = “core_clk”, “iface_clk”;
  clocks = <&clock_gcc clock_gcc_blsp1_uart2_apps_clk>,
       <&clock_gcc clk_gcc_blsp1_ahb_clk>;
  /* Using Pinctrl settings */
  pinctrl-names = "default";
  pinctrl-0 = <&lsuart_default>;

};
```

For detailed information, refer to the device tree documentation located at `/kernel/Documentation/devicetree/bindings/tty/serial/msm_serial.txt`.

2. Set the Pinctrl settings.
   a. Open the `.dtsi` file located at:

      `kernel/arch/arm/boot/dts/qcom/<chipset>-pinctrl.dtsi`

   b. Update the pin settings.

```
uart2-default {
      qcom,pins = <&gp 4>, <&gp 5>;
      qcom,num-grp-pins = <2>;
      qcom,pin-func = <2>;
      label = "uart2-default";
      lsuart_default: default {
                     drive-strength = <2>;
                     bias-pull-down;
               };
};
```

   From kernel 3.14 onwards that MSM8996 is using, pinctrl framework changes a bit. Refer to the following demo to configure the UART GPIOs.

```
&soc {
        tlmm: pinctrl@01010000 {
//snip

                lsuart_default: uart_console_active {
                        pmx-uartconsole {
                                pins = "gpio4", "gpio5";
                                function = "blsp_uart8"; //For how to
get the function name, please check the pinctrl driver code at
kernel/drivers/pinctrl/qcom/pinctrl-msm8996.c
                                drive-strength = <2>;
                                bias-disable;
                        };
                };
};
```

3.3.2 Optional configuration changes

After basic UART functionality is verified, you can enhance UART_DM functionality by configuring runtime GPIO and preventing system suspend.

3.3.2.1 Prevent system suspend

If required when the UART is in operation, the UART driver can prevent system suspend by automatically holding a wakelock.

3. Update the device tree. Open the device tree file located at:

   `/kernel/arch/arm/boot/dts/qcom/<chipset>-cdp.dtsi`

4. Add the use-pm node.

```
//Add following additional nodes to enable wakelock
uart2: uart@f991e000 { //0xF991E000 is the UART_DM Base address for
BLSP1_UART1
  qcom,use-pm; //Whenever port open wakelock will be held
```

5. Confirm that the UART driver is holding the wakelock.
   a. Open the UART port.

```
adb shell
cat /dev/ttyHSL#
```

   b. Dump the wake-up sources.

```
cat /sys/kernel/debug/wakeup_sources

msm_serial_hslite_port_open     2 2 0 0 1430 - Confirm
active_since != 0
```

6. Close the UART port. Confirm that `active_since` returns to zero.

For more information, see `/kernel/Documentation/devicetree/bindings/tty/serial/msm_serial.txt`.

3.3.3 Debug low-speed UART

1. Check the UART registration. Ensure that the UART is properly registered with the TTY stack.
2. Run the following commands:

```
adb shell -> start a new shell
ls /dev/ttyHSL* -> Make sure UART is properly registered
```

   If you do not see your device, check your code modification to ensure that all the information is defined and correct.

3. Check the bus scale registration. Ensure that the UART is properly registered with the bus scale driver.
   a. Run the following commands:

```
adb shell
mount -t debugfs none /sys/kernel/debug      -> mount debug fs
cat /dev/ttyHSL#-> Open the UART port
```

   b. Go to the bus scale directory.

```
cd /sys/kernel/debug/msm-bus-dbg/client-data
ls
```

c. Confirm that the name that was put on msm-bus is there, for example, serial_uart2.

d. Cat client_name, for example:

```
cat serial_uart2

Output: Confirm curr = 1, and rest of values.
    curr   : 1
    masters: 86
    slaves : 512
    ab     : 500000
    ib     : 800000
```

If you do not see your device, check your code modification to ensure that all of the information is defined and correct.

4. Check the internal loopback. Run the following commands to enable loopback:

```
adb shell
mount -t debugfs none /sys/kernel/debug -> mount debug fs
cd /sys/kernel/debug/msm_serial_hsl     -> directory for Low Speed UART
echo 1 > loopback.#                     -> enable loopback. # = device #
cat loopback.#                          -> make sure returns 1
```

5. Open another shell to dump the UART Rx data.

```
adb shell
cat /dev/ttyHSL#  ->Dump any data UART Receive
```

6. Transmit some test data through a separate shell.

```
adb shell
echo "This Document Is Very Much Helpful" > /dev/ttyHSL# ->Transfer data
```

- If the loopback works:
  - Test message loop appears continuously in the command shell until you exit the cat program. This is because of the internal loopback and how the cat program opens the UART.
  - It is safe to assume that the UART is properly configured and only the GPIO settings need to be confirmed.
- If loopback does not work:

i Ensure that the UART is still in the Active state. Open the UART from the shell:
```
adb shell
cat /dev/ttyHSL#   ->Dump any data UART Receive
```

ii Check the clock settings.

iii Measure the clocks from the debug-fs command.

- Make sure the PNoC clock is running.
  `cat /sys/kernel/debug/clk/pnoc_clk/measure`
- Measure the BLSP AHB clock.

  `label: gcc_blsp1:2_ahb_clk`

  For example, `cat /sys/kernel/debug/clk/gcc_blsp1_ahb_clk/measure`
- Measure the UART core clock.

  `label: gcc_blsp1:2_uart1:6_apps_clk`

  For example, `cat /sys/kernel/debug/clk/gcc_blsp1_uart3_apps_clk/measure`

- Loopback works, but there is no signal output to check the GPIO settings. For instructions, see section *UART→Configure LK UART→Code changes*.

3.4 Configure kernel high-speed UART

This high speed UART driver (kernel/drivers/tty/serial/msm_serial_hs.c) is designed for high-speed, large data transfers, such as Bluetooth communication. The hardware UART core is the same as the low-speed UART core. The biggest difference between this high-speed UART driver and the low-speed uart driver described in 3.3 is BAM pipes are enabled for high-speed UART.

3.4.1 Code changes

NOTE: Numerous changes were made in this section.

Along with the files mentioned in section *UART→Configure kernel low-speed UART→Code changes*, the following file is also used to configure high-speed UART.

File type	Description

TrustZone	The TrustZone header file is located here:
	`/trustzone_images/core/hwengines/bam/<chip>/bamtgtcfgdata_tz.h`
	Or
	`/trustzone_images/core/hwengines/bam/<chip>/bamtgtcfgdata.h`
	Where <*chip*> corresponds to the 4-digit chip number, for example:
	`/trustzone_images/core/hwengines/bam/8994/bamtgtcfgdata_tz.h`

The following procedure describes how to configure BLSP2_UART2 (BLSP8) as a high-speed UART in the MSM8994 platform:

1. Create a device tree node.
   a. Open the device tree file located at:

```
/kernel/arch/arm/boot/dts/qcom/msm8994.dtsi
```

   b. Modify the configuration. The elements described in the following example are the minimum requirements.

```
blsp2_uart2: uart@f995e000 { //0xF995E000 is the UART_DM Base Address
compatible = "qcom,msm-hsuart-v14"; //manufacture, model of serial driver
status = "ok"; //Status OK enables it

/* First Row UART_DM Base and Size always 0x1000
   Second Row is BAM address, size always 0x19000
   For BLSP1 UART0:5 Bam Address = 0xF9904000
   For BLSP2 UART0:5 Bam Address = 0xF9944000
*/
reg = <0xf995e000 0x1000>,
      <0xf9944000 0x19000>; //For 8916/8939/8909, size is 0x23000
reg-names = reg-names = "core_mem", "bam_mem"; //Keep the same names


 /* Interrupt map
    First Field: Interrupt index within interrupt map
    Second Field: Interrupt parent
    Third Field: 0 SPI interrupt (Shared Peripheral Interrupt)
    Fourth Field: Interrupt #
    Fifth field: Trigger type, keep 0
    See also:/kernel/Documentation/devicetree/bindings/arm/gic.txt
    And
    /kernel/Documentation/devicetree/bindings/arm/gic-v3.txt
 */
/* ATTENTION:
   In newer kernel, there is a GICv3 driver change which requests
   one more parameters of the interrupt map of intc.
  If we encounter following errors during driver init stage:
   ********************************************************
   msm_serial_hs *.uart: Error -6, invalid bam irq resources.
   ********************************************************
  Except to check whether the bam irq is correctly configured,
  we need to check whether the new interrupt map should be used:
   interrupt-map = <0 &intc 0 0 114 0 //add 0 to adapt the GICV3 change
                    1 &intc 0 0 239 0 //add 0 to adapt the GICV3 change
                    2 &msm_gpio 46 0>;
 */


interrupt-names = "core_irq", "bam_irq", "wakeup_irq";
#address-cells = <0>;
interrupt-parent = <&blsp2_uart2>;
interrupts = <0 1 2>;
#interrupt-cells = <1>;
interrupt-map-mask = <0xffffffff>;
interrupt-map = <0 &intc 0 114 0
                 1 &intc 0 239 0
```

```
                    2 &msm_gpio 46 0>;


        qcom,bam-tx-ep-pipe-index = <2>; //BAM Consumer Pipe
        qcom,bam-rx-ep-pipe-index = <3>; //BAM Producer Pipe
        qcom,master-id = <84>;

        /* Documentation/devicetree/bindings/arm/msm/msm_bus.txt
           name = Client name Keep same name as interface
           num-cases = # of use-cases, keep as 2.
           active-only = false (support dual context active   sleep)
           num-paths = 1 for Master-Slave Pairs.
           vectors-KBps = master-id, slave-id, bandwidth settings..
           master-id = 86 for BLSP1
           84 for BLSP2
           EBI CH0 slave-id = 512
            bandwidth settings: Keep @ below settings "0 0", "500 800"
         */
         qcom,msm-bus,name = "buart8";
         qcom,msm-bus,num-cases = <2>;
         qcom,msm-bus,num-paths = <1>;
         qcom,msm-bus,vectors-KBps =
                                    <86 512 0 0>,
                                    <86 512 500 800>;

         /* Select nodes for AHB and Core clocks
            In clock regime UART clocks are labeled #1 to #6. So BLSP2 UART2 is
            clk_gcc_blsp2_uart2_apps_clk
          */
          clock-names = "core_clk", "iface_clk";
          clocks = <&clock_gcc clock_gcc_blsp2_uart2_apps_clk>,
                   <&clock_gcc clk_gcc_blsp2_ahb_clk>;

           /* Using Pinctrl settings */
           pinctrl-names = "default", "sleep”;
           pinctrl-0 = <&hsuart_active>;
           pinctrl-1 = <&hsuart_sleep>;

         };
```

Additional information	Location
Device tree	/kernel/Documentation/devicetree/bindings/tty/serial/msm_serial_hs.txt
UART_DM interrupt values	/kernel/Documentation/devicetree/bindings/arm/gic.txt
Device tree bindings	/kernel/Documentation/devicetree/bindings/arm/msm/msm_bus.txt
Master ID	/kernel/arch/arm/boot/dts/<chip>-bus.dtsi
Pin control	/kernel/Documentation/devicetree/bindings/pinctrl/msm-pinctrl.txt

2. Set the Pinctrl settings.
   a. Please refer to the chipset device specification to check BLSP gpio assignment, and to the chipset software interface to check BLSP gpio configuration.
   b. Open the `.dtsi` file located at:

kernel/arch/arm/boot/dts/qcom/<*chipset*>-pinctrl.dtsi

c. Modify the pin control settings as shown in the following example. For more information, refer to pin control documentation located at /kernel/Documentation/devicetree/bindings/pinctrl/msm-pinctrl.txt.

```
&soc {
  tlmm_pinmux: pinctrl@fd510000 {

//snip
blsp2_uart2_active {
      qcom,pins = <&gp 45>, <&gp 46>, <&gp 47>, <&gp 48>;
      qcom,num-grp-pins = <4>;
      qcom,pin-func = <2>;// OEM must get the right gpio func number from
the software interface: TLMM chapter!
      label = "blsp2_uart2_active";
      hsuart_active: active {
         drive-strength = <16>;
         bias-disable;
      };
};
blsp2_uart2_sleep {
      qcom,pins = <&gp 45>, <&gp 46>, <&gp 47>, <&gp 48>;
      qcom,num-grp-pins = <4>;
      qcom,pin-func = <0>;
      label = "blsp2_uart2_sleep";
      hsuart_sleep: sleep {
         drive-strength = <2>;
         bias-disable;
      };
};
```

From kernel 3.14 onwards that MSM8996 is using, pinctrl framework changes a bit. Refer to the following demo to configure the UART GPIOs:

```
&soc {
   tlmm: pinctrl@01010000 {
                compatible = "qcom,msm8996-pinctrl";

//snip
                uart2-active: uart2-active {
                        mux {
                                pins = "gpio4", "gpio5", "gpio6", "gpio7";
                                function = " blsp_uart8"; //For how to get
the function name, please check the pinctrl driver code at
kernel/drivers/pinctrl/qcom/pinctrl-msm8996.c
                        };

                        config {
                                pins = "gpio4", "gpio5", "gpio6", "gpio7";
                                drive-strength = <2>;
                                bias-disable;
                        };
                };

                uart2-sleep: uart2-sleep {
                        config {
                                pins = "gpio4", "gpio5", "gpio6", "gpio7";
                                drive-strength = <2>;
                                bias-disable;
                        };
                };
…
```

3. Modify TrustZone. Each BLSP can be used by any subsystem such as modem and LPASS. It is important to give ownership of the BAM pipes apps processor.
   a. Open the following file:

      `/trustzone_images/core/hwengines/bam/<chip>/bamtgtcfgdata_tz.h`

      Or

      `/trustzone_images/core/hwengines/bam/<chip>/bamtgtcfgdata.h`

b. Allocate BLSP BAM pipes with the applications core.

```
/*
  bam_tgt_blsp1_secconfig = BLSP1 Core
  bam_tgt_blsp2_secconfig = BLSP2 Core
  Each bit you set represent Pipe #.
    For example Bit 0 = Pipe # 0
                Bit 5 = Pipe # 5

  Any bit set on TZBSP_VMID_AP is owned by APPs
*/
bam_sec_config_type bam_tgt_blsp1_secconfig =
{
    { //BLSP1_UART1 is pipe 2, 3. Set Bits 2 and 3 to APPS
        {0x00C0FFFF, TZBSP_VMID_AP}, //Bit 2,3 should be set
        {0x003F0000, TZBSP_VMID_LPASS } //Bit 2,3 should be cleared
    }
};
bam_sec_config_type bam_tgt_blsp2_secconfig =
{
    {
        {0x003C0FFF, TZBSP_VMID_AP},
        {0x00C3F000, TZBSP_VMID_LPASS}
    }
};
```

3.4.2 Debug high-speed UART

1. Check the registration. Ensure that the UART is properly registered with the TTY stack by running the following commands:

```
adb shell -> start a new shell
ls /dev/ttyHS* -> Make sure UART is properly registered
```

   If the device does not appear, check your code modification to ensure that all of the information is defined and correct.

2. Check the internal loopback.

   a. Run the following commands to enable loopback:

```
adb shell
mount -t debugfs none /sys/kernel/debug  -> mount debug fs
cd /sys/kernel/debug/msm_serial_hs       -> directory for High Speed
UART
echo 1 > loopback.#                      -> enable loopback. # is
device #
cat loopback.#                           -> make sure returns 1
```

   b. Open another shell to dump the UART Rx data.

```
adb shell
cat /dev/ttyHS#  ->Dump any data UART Receive
```

c. Transmit some test data through a separate shell.

```
adb shell
echo "This Is A Helpful Document" > /dev/ttyHS# ->Transfer data
```

If loopback works:

- Your test message loops continuously in the command shell until you exit the cat program. This is because of the internal loopback and how the cat program opens the UART.
- UART is properly configured and only the GPIO settings need to be confirmed.

If loopback works but there is no output:

- Check the GPIO settings as described in section *UART→Configure LK UART→Code changes*.

If loopback does not work:

d. Ensure that the apps processor can access the designated pipe.

Table 3-8 BLSP UART BAM pipe physical address

BLSP pipe number	PIPE_TRUST_REG	PIPE_CTRL_REG
BLSP1_P 0	0xF9905030	0xF9905000
BLSP1_P 1	0xF9906030	0xF9906000
BLSP1_P 2	0xF9907030	0xF9907000
BLSP1_P 3	0xF9908030	0xF9908000
BLSP1_P 4	0xF9909030	0xF9909000
BLSP1_P 5	0xF990A030	0xF990A000
BLSP1_P 6	0xF990B030	0xF990B000
BLSP1_P 7	0xF990C030	0xF990C000
BLSP1_P 8	0xF990D030	0xF990D000
BLSP1_P 9	0xF990E030	0xF990E000
BLSP1_P 10	0xF990F030	0xF990F000
BLSP1_P 11	0xF9910030	0xF9910000
BLSP2_P 0	0xF9945030	0xF9945000
BLSP2_P 1	0xF9946030	0xF9906000
BLSP2_P 2	0xF9947030	0xF9947000
BLSP2_P 3	0xF9948030	0xF9908000
BLSP2_P 4	0xF9949030	0xF9949000
BLSP2_P 5	0xF994A030	0xF990A000
BLSP2_P 6	0xF994B030	0xF994B000
BLSP2_P 7	0xF994C030	0xF990C000
BLSP2_P 8	0xF994D030	0xF994D000
BLSP2_P 9	0xF994E030	0xF990E000
BLSP2_P 10	0xF994F030	0xF994F000
BLSP2_P 11	0xF9950030	0xF9950000

The MSM8916, MSM8936, MSM8909, and MDM9x35 chipsets contain a single BLSP core, so BLSP2_Pipe0:11 is applicable only for the MSM8994 and MSM8992 chipset.

e. After the system boots up, attach a JTAG to the RPM and Krait cores (t32 cmd: sys.m.a}).

f. Break all of the Krait cores (t32 cmd: b}).

g. Turn on the BLSP_AHB_CLK by ensuring that bits 17 and 15 are set for the APCS_CLOCK_BRANCH_ENA_VOTE register (APCS_CLOCK_BRANCH_ENA_VOTE = 0xFC401484).

NOTE: Only bit 17 is required for the MSM8916, MSM8936, MSM8909, and MDM9x35 chipsets.

h. Ensure that BLSP_PIPE_TRUST_REG is set to zero for the apps processor. (Read using Secure mode.)

```
D AZ:0xF9907030} – BLSP1 pipe 2
D AZ:0xF9908030} – BLSP1 pipe 3
```

i. Ensure that you can read/write to PIPE_CTRL_REG in Non secure mode.

```
D.S A:0x0:0xF9907000 %LE %LONG 0xABCD} – Writing pipe 2
D.S A:0x0:0xF9908000 %LE %LONG 0xABCD} – Writing pipe 3
data.in A:0xF9907000 /long} – Reading pipe 2
data.in A:0xF9908000 /long} – Reading pipe 3
```

j. Check the TrustZone modifications.

3. Check the clock settings.

a. Ensure that the UART is still in Active state.

b. Open the UART from the shell:

```
adb shell
cat /dev/ttyHS#  ->Dump any data UART Receive
```

For instructions on checking the clock settings, see section *UART→Configure LK UART→Code changes.*

3.5 Code walkthrough – High-speed UART driver

This section explains the details of implementing a high-speed UART driver for debugging or modifications.

3.5.1 Probing

If UARTs are defined in the Device Tree, the msm_hs_probe() function is called, as shown in the following call flow:

```
msm_serial_hs_init() ->
  platform_driver_register(&msm_serial_hs_platform_driver) ->
    drv = &msm_serial_hs_platform_driver.driver;
    drv->bus = &platform_bus_type;
    driver_register (drv) ->
      bus_add_driver(drv) ->
        driver_attach(drv) ->
          bus_for_each_dev(drv->bus,..., drv,..)
            Iterate thru bus list of devices (bus->p->klist_devices)
            driver_attach(drv, dev) ->
              platform_match() ->
                Checks if the current dev match drv by comparing
                drv.of_match_table with dev.of_node.  If match
                found calls driver_probe_device
                driver_probe_device(drv, dev) ->
                  platform_drv_probe(..) ->
                    msm_hs_probe()
```

Table 3-9 Resources required for UART registration

Resource	Description
msm_hs_dt_to_pdata	Parses device tree nodes
msm_bus_cl_get_pdata	Parses device tree for bus scale nodes
q_uart_port[id]	Stores the parsed data
Device tree	
core_mem	UART base address
bam_mem	BLSP BAM base address
qcom,bam-rx-ep-pipe-index	BAM Rx pipe index
qcom,bam-tx-ep-pipe-index	BAM Tx pipe index
core_irq	UART peripheral IRQ
bam_irq	BLSP BAM IRQ
Clock table	
core_clk	UART core clock
iface_clk	BUS interface clock

BUS scale information is parsed by the bus scale driver

3.5.1.1 Registration with the SPS driver

During a probe, the UART driver registers BLSP BAM with the SPS/BAM driver, as shown in the following call flow.

```
msm_hs_probe()->
  msm_hs_sps_init()-->
    sps_phy2n()-->sps_register_bam_device()
    msm_hs_sps_init_ep_conn(Producer Info)
```

```
msm_hs_sps_init_ep_conn(Consumer Info)
```

The `msm_hs_probe()` function performs the following actions:

- Calls sps_phy2h() to check if the current BLSP BAM is already registered with the SPS driver. If the current BAM is registered, returns the handler for the BAM.
- Calls sps_register_bam_device() to register the BLSP BAM with the SPS driver if the BAM is not registered.
- Calls msm_hs_sps_init_ep_conn() to initialize BAM connection information:
  - Allocates memory for descriptor FIFO (sps_config to desc.base, sps_config to desc.size)
  - The Event mode is a function callback:
    - For UART Rx operations, the callback is called when the descriptor is complete.
    - For UART Tx operations, the callback is called when the End-Of-Transfer (EOT) bit is set.

3.5.1.2 UART port registration

The UART driver registers the current UART port with the Linux TTY stack, as shown in the following call flow.

```
msm_hs_probe()->
  uart_add_one_port()->
    uart_configure_port()->
      msm_hs_config_port()-Sets uart->type to PORT_MSM
      msm_hs_set_mctrl_locked()-Set RFR High (not accepting data)
    <-
    tty_register_device()- Registers with tty framework
```

3.5.2 Port open

The following call flow shows critical events that occur when the client opens a UART port.

```
tty_open()->
  uart_open()->
    uart_startup()->
      uart_port_startup()->
        msm_hs_startup()-->
          msm_hs_resource_vote()()-Turns on clks
          msm_hs_config_uart_gpios()-request GPIOs
          msm_hs_spsconnect_tx/rx()
            sps_connect()
            sps_register_event()
          <--
          Configure UART Hardware
            msm_hs_start_rx_locked()
```

```
                sps_transfer_one()
        <------
          uart_change_speed()-->
            msm_hs_set_termios()-->
              msm_hs_set_bps_locked()
            <--
              sps_disconnect()
            <--
              msm_hs_spsconnect_rx()
            <--
              msm_serial_hs_rx_work()-->
                msm_hs_start_rx_locked()
<--------------
```

The uart_open() function performs the following actions:

- Increments port->count.
- If a port is not initialized (port->flags and ASYNC_INITIALIZED):
  - Allocates and clears a Tx buffer (uart_state->xmit.buf).
  - Calls msm_hs_startup().

The msm_hs_startup() function initializes the low-level UART core:

- Maps the Tx buffer to be a DMA capable buffer.
- Turns on all necessary clocks, including the bus scale request.
- If runtime GPIO configuration is enabled, requests the GPIOs.
- Initializes the BAM connection.
- Initializes the UART hardware:
  - UART_DM_MR1 – Sets the RFR watermark to FIFOSIZE-16
  - ART_DM_IPR – Sets RXSTALE interrupt counter to 0x1F
  - UART_DM_DMEN – Enables the Tx/Rx BAM
  - UART_DM_CR – Resets the transmitter
  - UART_DM_CR – Resets the receiver
  - UART_DM_CR – Clears the error status
  - UART_DM_CR – Clears the Break Change interrupt status bit
  - UART_DM_CR – Clears the Stale interrupt status bit
  - ART_DM_CR – Clears the CTS input change interrupt status bit
  - UART_DM_CR – Asserts the RFR signal
  - UART_DM_CR – Enables the receiver
  - UART_DM_CR – Turns on the transmitter
  - UART_DM_TFWR – Sets the Tx FIFO watermark to zero

- Enables the interrupt, and registers the ISR handler:
  - If the Wake Up interrupt is supported and enabled, registers the ISR handler but disables the interrupt.
- Enables Rx transfer (msm_hs_start_rx_locked()):
  - Configures the UART hardware:
    - UART_DM_CR – Clears the Stale interrupt
    - UART_DM_RX – Programs the maximum transfer length (UARTDM_RX_BUF_SIZE)
    - UART_DM_CR – Enables the Stale Event mechanism
    - UART_DM_DMEN – Enables Rx BAM mode
    - UART_DM_IMR – Enables the Stale Event interrupt
    - UART_DM_RX_TRANS_CTRL – Enables automatic re-transfer
    - UART_DM_CR – Initializes the BAM producer sideband signals
  - Queues a BAM descriptor, and initiates a transfer.

The msm_hs_set_termios() function performs the following actions:

- Disables UART interrupts and Rx BAM mode:
  - UART_DM_IMR – Sets to 0
  - UART_DM_DMEN – Clears the RX_BAM_EN bit
- Sets UART clock rates via msm_hs_set_bps_locked().
- Programs the UART hardware:
  - UART_DM_MR1, UART_DM_MR2 – For parity, flow controls, etc.
  - UART_DM_CR – Resets the receiver
  - UART_DM_CR – Resets the transmitter
- Disconnects from the SPS driver (sps_disconnect()).
- Reconnects the producer pipe with the SPS function (msm_hs_spsconnect_rx()).
- msm_serial_hs_rx_work():
  - Enables an Rx transfer via msm_hs_start_rx_locked().

3.5.3 Power management

The high speed UART driver defines Power management APIs as follows:

```
static const struct dev_pm_ops msm_hs_dev_pm_ops = {
   .runtime_suspend = msm_hs_runtime_suspend,
   .runtime_resume = msm_hs_runtime_resume,
   .runtime_idle = NULL,
   .suspend_noirq = msm_hs_pm_sys_suspend_noirq,
   .resume_noirq = msm_hs_pm_sys_resume_noirq,
};
```

In msm_hs_pm_sys_suspend_noirq(),

1. Clocks are turned OFF.
2. Core IRQ is disabled.
3. Wakeup IRQ, flow control is enabled if Out of Band Sleep not set
4. BAM pipes are disconnected
5. Runtime PM Framework is notified of the suspend state.

The driver maintains 3 power states:

```
MSM_HS_PM_ACTIVE – if it is in Active state i.e. all clocks are ON
MSM_HS_PM_SUSPENDED – if driver is Runtime suspend state
MSM_HS_PM_SYS_SUSPENDED – if driver is in System suspend state
```

3.5.3.1 In Band and Out Band Sleep modes

UART driver defines 2 sleep modes:

1. In Band Sleep: This suggests UART's wakeup IRQ(RX line) is enabled and RFR line asserted when it goes in suspend state. So that the UART client can wake it up by sending some data on RX line.

   This mode is enabled by the following DTS entries in UART node:

```
                interrupt-names = "core_irq", "bam_irq", "wakeup_irq";
//add "wakeup_irq" to the other IRQs list
                #address-cells = <0>;
                interrupt-parent = <&blsp2_uart2>;
                interrupts = <0 1 2>;
                #interrupt-cells = <1>;
                interrupt-map-mask = <0xffffffff>;
                interrupt-map = <0 &intc 0 114 0
                                 1 &intc 0 239 0
                                 2 &msm_gpio 46 0>; //RX GPIO number is
set as Wakeup IRQ
                qcom,rx-char-to-inject = <0xFD>; //This character is
injected on TX when wakeup IRQ received
                qcom,inject-rx-on-wakeup; //This enables the above
character injection
```

2. Out of Band Sleep: This suggests the UART client will explicitly call UART clock ON API to turn ON the clocks, before doing a transfer.

   This mode is enabled by the following DTS entry:

```
qcom,msm-obs;
```

3.5.3.2 Methods to control UART clocks

HS UART clocks can be turned ON/OFF in either of the following ways:

sys_fs call to control HS UART clock from user space

```
echo 0|1 > /sys/devices/BaseAddress.uart/clock}: ex: turn off/on clock
or
echo 0|1 > /sys/devices/soc.0/BaseAddress.uart/clock}: ex: turn off/on
clock
```

The following example is based upon BLSP7(BLSP2 UART0):F995d000 being configured as a high speed UART.

```
echo 0 > /sys/devices/f995d000.uart/clock → turn off the clock
echo 1 > /sys/devices/f995d000.uart/clock → turn on the clock
or
echo 0 > /sys/devices/soc.0/f995d000.uart/clock → turn off the clock
echo 1 > /sys/devices/soc.0/f995d000.uart/clock → turn on the clock
```

Kernel API

```
msm_hs_get_uart_port}, msm_hs_request_clock_on|off}
```

Example usage

```
/* Get the UART Port with port ID */
struct uart_port *port = msm_hs_get_uart_port(0);
/* Request turn off Clocks */
msm_hs_request_clock_off(port);
/* Request turn on clock */
msm_hs_request_clock_on(port);
```

IOCTL from the user space

```
IOCTL cmd
MSM_ENABLE_UART_CLOCK -request clk on
MSM_DISABLE_UART_CLOCK - request clk off
MSM_GET_UART_CLOCK_STATUS - get current status
```

After turning off the clocks, it is important that no UART functions are called before the clocks are turned back on, including the UART close function.

3.5.4 Port close

The following call flow shows critical events that occur when the client closes the UART port.

```
tty_release()-->
  uart_close()-->
    tty_port_close_start()
  <--
    msm_hs_stop_rx_locked()
  <--
    uart_wait_until_sent()-->
      msm_hs_tx_empty() returns UART_DM_SR    TXEMT
  <---
    uart_shutdown()-->
     uart_update_mctrl()-->
       msm_hs_set_mctrl_locked()
     <--
     uart_port_shutdown()-->
       msm_hs_shutdown()
<------

*Can run anytime after msm_hs_stop_rx_locked()
while uart_close()
hsuart_disconnect_rx_endpoint_work()-->
  sps_disconnect()--Disconnect/disable BAM connection
  and set msm_uport->rx.flush = FLUSH_SHUTDOWN;
<--
```

The `uart_close()` function performs the following actions:

- Calls tty_port_close_start() to decrement port->counts.
- Calls msm_hs_stop_rx_locked():
  - Clears the RX_BAM_ENABLE bit in UART_DM_DMEN to disable the Rx BAM interface.
  - Sets the rx.flush state to FLUSH_STOP.
  - Schedules the BAM work queue to be disconnected (hsuart_disconnect_rx_endpoint_work()).
- uart_wait_until_sent():
  - Continuously polls by calling msm_hs_tx_empty() until the UART_DM_SR[TXEMT] bit is set by the hardware.
- Calls uart_shutdown():
  - Sets the TTY_IO_ERROR bit to tty->flags.
  - Clears the ASYNCB_INITIALIZED bit to port->flags.
  - De-asserts RFR, and disables the Auto Ready to Receive bit.

- msm_hs_shutdown():
  - If a Tx is pending (which should not occur), disables and disconnects by calling sps_disconnect().
  - Waits until the hsuart_disconnect_rx_endpoint_work() function runs, and then sets rx.flush to FLUSH_SHUTDOWN.
  - Configures the UART hardware:
    - UART_DM_CR – Disables the transmitter
    - UART_DM_CR – Disables the receiver
    - UART_DM_IMR – Clears the interrupt mask register
  - Turns off the clocks, and sets clk_state to MSM_HS_CLK_PORT_OFF.
  - Frees IRQ resources.
  - Releases any GPIO resources.
- Frees allocated memory.
- Flushes the TTY and LDISC buffers.

4 I2C

This chapter describes the Inter-Integrated Circuit (I2C) and explains how to configure it in the kernel.

4.1 Hardware overview

Qualcomm Universal Serial Engine (QUP)

QUP provides a general purpose data path engine to support multiple mini cores. Each mini core implements protocol-specific logic. The common FIFO provides a consistent system IO buffer and system DMA model across widely varying external interface types. For example, one pair of FIFO buffers can support SPI and I2C mini cores independently.

Supported mini cores are:

- I2C
- SPI (see Chapter 5)

I2C core

On MSM8994, the Linux I2C driver supports Fast mode plus (up to 1 MHz). The following key features have been added:

- Duty-cycle control
- BAM integration
- Support for I2C tag version 2

The following features are not supported:

- Multi Master mode
- 10-bit slave address, and also the 10-bit extend address (for example, 1111 0XX) listed in I2C specification cannot be used by any slave device.
- HS mode (3.4 MHz clock frequency).

QUP I2C configuration parameters

In hardware documentations, BLSPs are identified as BLSP [1:12].

In the software documentation, each QUP is identified by a BLSP core (1 or 2) and QUP core (0 to 5).

Table 4-1 QUP physical address, IRQ numbers, Kernel I2C clock name, consumer, producer pipes, BLSP_BAM physical address, BAM IRQ number for MSM8994 and MSM8992

BLSP hardware ID	QUP core	Physical address (QUP_BASE_ADDRESS), size	IRQ #	Bus master ID	Kernel I2C clock name	Consumer, producer pipes	BLSP_BAM physical address, size, IRQ number
BLSP 1	BLSP1 QUP0	0xF9923000.0x1000	95	86	clk_gcc_blsp1_qup1_i2c_apps_clk	12, 13	0xF9904000, 0x19000, 238
BLSP 2	BLSP1 QUP1	0xF9924000.0x1000	96	86	clk_gcc_blsp1_qup2_i2c_apps_clk	14, 15	0xF9904000, 0x19000, 238
BLSP 3	BLSP1 QUP2	0xF9925000.0x1000	97	86	clk_gcc_blsp1_qup3_i2c_apps_clk	16, 17	0xF9904000, 0x19000, 238
BLSP 4	BLSP1 QUP3	0xF9926000.0x1000	98	86	clk_gcc_blsp1_qup4_i2c_apps_clk	18, 19	0xF9904000, 0x19000, 238
BLSP 5	BLSP1 QUP4	0xF9927000.0x1000	99	86	clk_gcc_blsp1_qup5_i2c_apps_clk	20, 21	0xF9904000, 0x19000, 238
BLSP 6	BLSP1 QUP5	0xF9928000.0x1000	100	86	clk_gcc_blsp1_qup6_i2c_apps_clk	22, 23	0xF9904000, 0x19000, 238
BLSP 7	BLSP2 QUP0	0xF9963000.0x1000	101	84	clk_gcc_blsp2_qup1_i2c_apps_clk	12, 13	0xF9944000, 0x19000, 239
BLSP 8	BLSP2 QUP1	0xF9964000.0x1000	102	84	clk_gcc_blsp2_qup2_i2c_apps_clk	14, 15	0xF9944000, 0x19000, 239
BLSP 9	BLSP2 QUP2	0xF9965000.0x1000	103	84	clk_gcc_blsp2_qup3_i2c_apps_clk	16, 17	0xF9944000, 0x19000, 239
BLSP10	BLSP2 QUP3	0xF9966000.0x1000	104	84	clk_gcc_blsp2_qup4_i2c_apps_clk	18, 19	0xF9944000, 0x19000, 239
BLSP11	BLSP2 QUP4	0xF9967000.0x1000	105	84	clk_gcc_blsp2_qup5_i2c_apps_clk	20, 21	0xF9944000, 0x19000, 239
BLSP12	BLSP2 QUP5	0xF9968000.0x1000	106	84	clk_gcc_blsp2_qup6_i2c_apps_clk	22, 23	0xF9944000, 0x19000, 239

Table 4-2 QUP physical address, IRQ numbers, Kernel I2C clock name, consumer, producer pipes, BLSP_BAM physical address, BAM IRQ number for MSM8916, MSM8936, MSM8939, and MSM8909

BLSP hardware ID	QUP core	Physical address (QUP_BASE_ADDRESS), size	IRQ #	Bus master ID	Kernel I2C clock name	Consumer, producer pipes	BLSP_BAM physical address, size, IRQ number
BLSP 1	BLSP1 QUP0	0x78B5000,0x1000	95	86	clk_gcc_blsp1_qup1_i2c_apps_clk	4, 5	0x7884000, 0x23000, 238
BLSP 2	BLSP1 QUP1	0x78B6000,0x1000	96	86	clk_gcc_blsp1_qup2_i2c_apps_clk	6, 7	0x7884000, 0x23000, 238
BLSP 3	BLSP1 QUP2	0x78B7000,0x1000	97	86	clk_gcc_blsp1_qup3_i2c_apps_clk	8, 9	0x7884000, 0x23000, 238
BLSP 4	BLSP1 QUP3	0x78B8000,0x1000	98	86	clk_gcc_blsp1_qup4_i2c_apps_clk	10, 11	0x7884000, 0x23000, 238
BLSP 5	BLSP1 QUP4	0x78B9000,0x1000	99	86	clk_gcc_blsp1_qup5_i2c_apps_clk	12, 13	0x7884000, 0x23000, 238
BLSP 6	BLSP1 QUP5	0x78BA000,0x1000	100	86	clk_gcc_blsp1_qup6_i2c_apps_clk	14, 15	0x7884000, 0x23000, 238

Table 4-3 QUP physical address, IRQ numbers, Kernel I2C clock name, consumer, producer pipes, BLSP_BAM physical address, BAM IRQ number for MSM8996

BLSP hardware ID	QUP core	Physical address (QUP_BASE_ADDRESS), size	IRQ #	Bus master ID	Kernel I2C clock name	Consumer, producer pipes	BLSP_BAM physical address, size, IRQ number
BLSP 1	BLSP1 QUP0	0x7575000,0x1000	95	86	clk_gcc_blsp1_qup1_i2c_apps_clk	12, 13	0x7544000, 0x2B000, 238
BLSP 2	BLSP1 QUP1	0x7576000,0x1000	96	86	clk_gcc_blsp1_qup2_i2c_apps_clk	14, 15	0x7544000, 0x2B000, 238
BLSP 3	BLSP1 QUP2	0x7577000,0x1000	97	86	clk_gcc_blsp1_qup3_i2c_apps_clk	16, 17	0x7544000, 0x2B000, 238
BLSP 4	BLSP1 QUP3	0x7578000,0x1000	98	86	clk_gcc_blsp1_qup4_i2c_apps_clk	18, 19	0x7544000, 0x2B000, 238
BLSP 5	BLSP1 QUP4	0x7579000,0x1000	99	86	clk_gcc_blsp1_qup5_i2c_apps_clk	20, 21	0x7544000, 0x2B000, 238
BLSP 6	BLSP1 QUP5	0x757A000,0x1000	100	86	clk_gcc_blsp1_qup6_i2c_apps_clk	22, 23	0x7544000, 0x2B000, 238
BLSP 7	BLSP2 QUP0	0x75B5000,0x1000	101	84	clk_gcc_blsp2_qup1_i2c_apps_clk	12, 13	0x7584000, 0x2B000, 239
BLSP 8	BLSP2 QUP1	0x75B6000,0x1000	102	84	clk_gcc_blsp2_qup2_i2c_apps_clk	14, 15	0x7584000, 0x2B000, 239
BLSP 9	BLSP2 QUP2	0x75B7000,0x1000	103	84	clk_gcc_blsp2_qup3_i2c_apps_clk	16, 17	0x7584000, 0x2B000, 239
BLSP10	BLSP2 QUP3	0x75B8000,0x1000	104	84	clk_gcc_blsp2_qup4_i2c_apps_clk	18, 19	0x7584000, 0x2B000, 239
BLSP11	BLSP2 QUP4	0x75B9000,0x1000	105	84	clk_gcc_blsp2_qup5_i2c_apps_clk	20, 21	0x7584000, 0x2B000, 239
BLSP12	BLSP2 QUP5	0x75BA000,0x1000	106	84	clk_gcc_blsp2_qup6_i2c_apps_clk	22, 23	0x7584000, 0x2B000,239

Table 4-4 QUP physical address, IRQ numbers, Kernel I2C clock name, consumer, producer pipes, BLSP_BAM physical address, BAM IRQ number for MSM8952, MSM8953, and MSM8956/76

NOTE: The following table has been updated.

BLSP hardware ID	QUP core	Physical address (QUP_BASE_ADDRESS), size	IRQ #	Bus master ID	Kernel I2C clock name	Consumer, producer pipes	BLSP_BAM physical address, size, IRQ number
BLSP 1	BLSP1 QUP0	0x78B50000,0x600	95	86	clk_gcc_blsp1_qup1_i2c_apps_clk	4, 5	0x7884000, 0x1F000, 238
BLSP 2	BLSP1 QUP1	0x78B6000,0x600	96	86	clk_gcc_blsp1_qup2_i2c_apps_clk	6, 7	0x7884000, 0x1F000, 238
BLSP 3	BLSP1 QUP2	0x78B7000,0x600	97	86	clk_gcc_blsp1_qup3_i2c_apps_clk	8, 9	0x7884000, 0x1F000, 238
BLSP 4	BLSP1 QUP3	0x78B8000,0x600	98	86	clk_gcc_blsp1_qup4_i2c_apps_clk	10, 11	0x7884000, 0x1F000, 238
BLSP 5	BLSP2 QUP0	0x7AF5000,0x600	299	84	clk_gcc_blsp2_qup1_i2c_apps_clk	4, 5	0x7AC4000, 0x1F000, 239
BLSP 6	BLSP2 QUP1	0x7AF6000,0x600	300	84	clk_gcc_blsp2_qup2_i2c_apps_clk	6, 7	0x7AC4000, 0x1F000, 239

BLSP 7	BLSP2 QUP2	0x7AF7000,0x600	301	84	clk_gcc_blsp2_qup3_i2c_apps_clk	8, 9	0x7AC4000, 0x1F000, 239
BLSP 8	BLSP2 QUP3	0x7AF8000,0x600	302	84	clk_gcc_blsp2_qup4_i2c_apps_clk	10, 11	0x7AC4000, 0x1F000, 239

Table 4-5 QUP physical address, IRQ numbers, Kernel I2C clock name, consumer, producer pipes, BLSP_BAM physical address, BAM IRQ number for MDM9x35

BLSP hardware ID	QUP core	Physical address (QUP_BASE_ADDRESS), size		IRQ #	Bus master ID	Kernel I2C clock name	Consumer, producer pipes	BLSP_BAM physical address, size, IRQ number
BLSP 1	BLSP1 QUP0	0xF9923000.0x1000		95	86	clk_gcc_blsp1_qup1_i2c_apps_clk	12, 13	0xF9904000, 0x19000, 238
BLSP 2	BLSP1 QUP1	0xF9924000.0x1000		96	86	clk_gcc_blsp1_qup2_i2c_apps_clk	14, 15	0xF9904000, 0x19000, 238
BLSP 3	BLSP1 QUP2	0xF9925000.0x1000		97	86	clk_gcc_blsp1_qup3_i2c_apps_clk	16, 17	0xF9904000, 0x19000, 238
BLSP 4	BLSP1 QUP3	0xF9926000.0x1000		98	86	clk_gcc_blsp1_qup4_i2c_apps_clk	18, 19	0xF9904000, 0x19000, 238
BLSP 5	BLSP1 QUP4	0xF9927000.0x1000		99	86	clk_gcc_blsp1_qup5_i2c_apps_clk	20, 21	0xF9904000, 0x19000, 238
BLSP 6	BLSP1 QUP5	0xF9928000.0x1000		100	86	clk_gcc_blsp1_qup6_i2c_apps_clk	22, 23	0xF9904000, 0x19000, 238

Table 4-6 QUP physical address, IRQ numbers, Kernel I2C clock name, consumer, producer pipes, BLSP_BAM physical address, BAM IRQ number for MDM9x40/9x45, and MDM9x50/9x55

BLSP hardware ID	QUP core	Physical address (QUP_BASE_ADDRESS), size	IRQ #	Bus master ID	Kernel I2C clock name	Consumer, producer pipes	BLSP_BAM physical address, size, IRQ number
BLSP 1	BLSP1 QUP0	0x78B5000,0x1000	95	86	clk_gcc_blsp1_qup1_i2c_apps_clk	8, 9	0x7884000, 0x23000, 238
BLSP 2	BLSP1 QUP1	0x78B6000,0x1000	96	86	clk_gcc_blsp1_qup2_i2c_apps_clk	10, 11	0x7884000, 0x23000, 238
BLSP 3	BLSP1 QUP2	0x78B7000,0x1000	97	86	clk_gcc_blsp1_qup3_i2c_apps_clk	12, 13	0x7884000, 0x23000, 238
BLSP 4	BLSP1 QUP3	0x78B8000,0x1000	98	86	clk_gcc_blsp1_qup4_i2c_apps_clk	14, 15	0x7884000, 0x23000, 238

Table 4-7 QUP physical address, IRQ numbers, Kernel I2C clock name, consumer, producer pipes, BLSP_BAM physical address, BAM IRQ number for MDM9x07

NOTE: The following table was added to this document revision.

BLSP hardware ID	QUP core	Physical address (QUP_BASE_ADDRESS), size	IRQ #	Bus master ID	Kernel I2C clock name	Consumer, producer pipes	BLSP_BAM physical address, size, IRQ number
BLSP 1	BLSP1 QUP0	0x78B5000,0x1000	95	86	clk_gcc_blsp1_qup1_i2c_apps_clk	12, 13	0x7884000, 0x23000, 238
BLSP 2	BLSP1 QUP1	0x78B6000,0x1000	96	86	clk_gcc_blsp1_qup2_i2c_apps_clk	14, 15	0x7884000, 0x23000, 238
BLSP 3	BLSP1 QUP2	0x78B7000,0x1000	97	86	clk_gcc_blsp1_qup3_i2c_apps_clk	16, 17	0x7884000, 0x23000, 238
BLSP 4	BLSP1 QUP3	0x78B8000,0x1000	98	86	clk_gcc_blsp1_qup4_i2c_apps_clk	18, 19	0x7884000, 0x23000, 238
BLSP 5	BLSP1 QUP4	0x78B9000,0x1000	99	86	clk_gcc_blsp1_qup5_i2c_apps_clk	20, 21	0x7884000, 0x23000, 238
BLSP 6	BLSP1 QUP5	0x78BA000,0x1000	100	86	clk_gcc_blsp1_qup6_i2c_apps_clk	22, 23	0x7884000, 0x23000, 238

4.2 Configure LK I2C

This section describes how to configure and use any of the available QUP cores in the chipset as an I2C device.

Because LK is designed as a single thread application currently, the I2C driver has some APIs designed as non-reentrant functions. If OEMs need more than one I2C masters in LK stage, please follow the sample code step 5 of Section 4.2.1.

4.2.1 Code changes

The following files are used to configure a QUP core as I2C in a Linux kernel:

```
/bootable/bootloader/lk/project/<chipset>.mk
/bootable/bootloader/lk/target/<chipset>/init.c
/bootable/bootloader/lk/platform/<chipset>/include/platform/iomap.h
/bootable/bootloader/lk/platform/<chipset>/acpuclock.c
/bootable/bootloader/lk/platform/<chipset>/<chipset>-clock.c
/bootable/bootloader/lk/platform/<chipset>/gpio.c
```

The following procedure uses the MSM8994 chipset for example purposes. Similar changes can be applied to other chipsets.

1. Enable the console shell to demonstrate I2C.
   a. Open the following file:

      ```
      Project_root/bootable/bootloader/lk/project/<chipset>.mk
      ```

   b. To demonstrate I2C, create an LK shell program using the serial port.

      ```
      MODULE +=app/shell
      ```

      This is for testing/demonstration purposes only and is not required for I2C.

   c. To test, connect the serial terminal to the device. After compiling is finished, flash the aboot and reboot the device into fastboot. The following message appears on the terminal:

      ```
      console_init: entry
      starting app shell
      entering main console loop
      ```

   d. Test the shell by entering **help** in the terminal program.

      ```
      Sample output:  command list:
               help            : this list
               test            : test the command processor
      ```

2. Create a test program. This is an optional process to demonstrate I2C functionality.

   a. Create a test application in `/bootable/bootloader/lk/app/tests/my_i2c_test.c`.

```
#include <ctype.h>
#include <debug.h>
#include <stdlib.h>
#include <printf.h>
#include <list.h>
#include <string.h>
#include <arch/ops.h>
#include <platform.h>
#include <platform/debug.h>
#include <kernel/thread.h>
#include <kernel/timer.h>

#ifdef WITH_LIB_CONSOLE
#include <lib/console.h>
static int cmd_i2c_test(int argc, const cmd_args *argv);

STATIC_COMMAND_START
      { "i2c_test", "i2c test cmd", &cmd_i2c_test },
STATIC_COMMAND_END(my_i2c_test);

static int cmd_i2c_test(int argc, const cmd_args *argv)
{
      printf("Entering i2c_test\n");
      return 0;
}

#endif
```

   b. Modify `/bootable/bootloader/lk/app/tests/rules.mk` to enable the test application.

```
LOCAL_DIR := $(GET_LOCAL_DIR)
INCLUDES += -I$(LOCAL_DIR)/include
OBJS += $(LOCAL_DIR)/my_i2c_test.o
```

   c. Modify `/bootable/bootloader/lk/project/<chipset>.mk` to compile the test application.

```
MODULES += app/tests
```

   d. Verify that the `i2c_test` command is available as part of the shell command.

```
cmd "help"
command list:
        help            : this list
        test            : test the command processor

        i2c_test        : i2c test cmd
cmd "i2c_test"
Entering i2c_test
```

3. Configure I2C bus in LK.

   a. Initialize the I2C bus. The following code sample is for the BLSP2 QUP4 and uses `my_i2c_test.c` as the client driver.

```
#include <i2c_qup.h>
#include <blsp_qup.h>
{
	struct qup_i2c_dev *dev;
	/*
	  1 arg: BLSP ID can be BLSP_ID_1 or BLSP_ID_2
	  2 arg: QUP ID can be QUP_ID_0:QUP_ID_5
	  3 arg: I2C CLK. should be 100KHZ, or 400KHz
	  4 arg: Source clock, should be set @ 19.2MHz
	 */
	    dev = qup_blsp_i2c_init(BLSP_ID_2, QUP_ID_4,
	    100000, 19200000);

	if(!dev){
		printf("Failed to initialize\n");
		return;
	}
}
```

   b. Configure the GPIO. Modify `/bootable/bootloader/lk/platform/<chipset>/gpio.c` and change the `gpio_config_blsp_i2c` function by adding the appropriate GPIO configuration for the correct BLSP configuration:

   Please refer to the chipset device specification to check BLSP gpio assignment, and to the chipset software interface to check BLSP gpio configuration.

```
void gpio_config_blsp_i2c(uint8_t blsp_id, uint8_t qup_id)
{
	if (blsp_id == BLSP_ID_2) {
		switch (qup_id) {
		case QUP_ID_4:
				/*
				1 arg: GPIO #
				2 arg: GPIO Function (Check GPIO MAP)
				3 arg: Doesn't matter
				4 arg: Should be GPIO_NO_PULL
				5 arg: Drive Strength: Lower the better
				6 arg: doesn't matter
				*/
			gpio_tlmm_config(83, 3, GPIO_OUTPUT, GPIO_NO_PULL,
					GPIO_6MA, GPIO_DISABLE);


			gpio_tlmm_config(84, 3, GPIO_OUTPUT, GPIO_NO_PULL,
					GPIO_6MA, GPIO_DISABLE);
		break;
```

c. Register a clock. Modify `/bootable/bootloader/lk/platform/<chipset>/msm8994-clock.c` and add the clock node and corresponding QUP clock.

```
static struct clk_lookup msm_clocks_<chip>[] =
{
  /**
  Add Clock node for BLSP_AHB_CLOCK
    For BLSP1 you would add:
        "blsp1_ahb_clk", gcc_blsp1_ahb_clk.c
    For BLSP2 you would add:
        "blsp2_ahb_clk", gcc_blsp2_ahb_clk.c
  */
  CLK_LOOKUP("blsp2_ahb_clk",          gcc_blsp2_ahb_clk.c),

  /**
  Add corresponding QUP Clock.  Clocks are indexed from 1 to 6.
  So QUP4 would refer to QUP5 in clock regime
  */
  CLK_LOOKUP("blsp2_qup5_i2c_apps_clk",
     gcc_blsp2_qup5_i2c_apps_clk.c),
```

d. Add the clock structure if it is not defined yet.

```
struct branch_clk gcc_blsp2_qup5_i2c_apps_clk = {
        /*
          .cbcr_reg value is defined on bootable/bootloader/
          lk/platform/<chipset>/include/platform/iomap.h
          If its not defined, get the value from
          /kernel/arch/arm/mach-msm/clock-<chip>.c
        */
       .cbcr_reg = BLSP2_QUP5_I2C_APPS_CBCR,
        /*
          .parent you can get from
          /kernel/arch/arm/mach-msm/clock-<chip>.c
        */
       .parent   = &cxo_clk_src.c,

       .c = {
             .dbg_name = "gcc_blsp2_qup5_i2c_apps_clk",
             .ops      = &clk_ops_branch,
       },
};
```

4. Test I2C transfer functionality.

```
void my_i2c_test()
{
..
      char buf[10];
      struct i2c_msg msg;

      //Create a msg header
      msg.addr = 0x52;
      msg.flags = I2C_M_RD;
      msg.len = 10;
      msg.buf = buf;

        //Transfer the data
      ret = qup_i2c_xfer(dev, &msg, 1);
}
```

5. Use more I2C masters in LK.

```
int ret;
uint8_t data_buf[] = { addr, val };
 /* Create a i2c_msg buffer, that is used to put the controller into write
          mode and then to write some data. */
 struct i2c_msg msg_buf[] = {
{I2C_SLAVE_ADDRESS, I2C_M_WR, 2, data_buf}
 };

i2c_dev = qup_blsp_i2c_init(BLSP_ID_1, QUP_ID_1, 100000, 19200000);
 if(!i2c_dev) {
                        dprintf(CRITICAL, "qup_blsp_i2c_init failed \n");
                        ASSERT(0);
               }
  ret = qup_i2c_xfer(i2c_dev, msg_buf, 1);
   if(ret < 0) {
                        dprintf(CRITICAL, "qup_i2c_xfer error %d\n", ret);
                         return ret;
                  }

             /* After finish the i2c transfer of master A, try to free the
resource and init the 2nd I2C master. */
             memset(i2c_dev, 0, sizeof(  qup_i2c_dev));
             qup_i2c_deinit(i2c_dev);

             /* Try to initialize i2c master 2 and use it */
            i2c_dev = qup_blsp_i2c_init(BLSP_ID_1, QUP_ID_2, 100000,
19200000);
 if(!i2c_dev) {
```

```
                        dprintf(CRITICAL, "qup_blsp_i2c_init failed \n");
                        ASSERT(0);
                }
  ret = qup_i2c_xfer(i2c_dev, msg_buf, 1);
   if(ret < 0) {
                        dprintf(CRITICAL, "qup_i2c_xfer error %d\n", ret);
                         return ret;
                    }
            /* After finish the i2c transfer of master A, try to free the
resource and init the 2nd I2C master.
            memset(i2c_dev, 0, sizeof(  qup_i2c_dev));
            qup_i2c_deinit(i2c_dev);
```

4.2.2 Test code

```
#include <i2c_qup.h>
#include <blsp_qup.h>
#include <board.h>


void my_i2c_test()
{
	struct qup_i2c_dev *dev;
	char buf[10];
	struct i2c_msg msg;
	int ret,i;
	  int soc_ver = board_soc_version(); //Get the CHIP version

	/*
	  1 arg: BLSP ID can be BLSP_ID_1 or BLSP_ID_2
	  2 arg: QUP ID can be QUP_ID_0:QUP_ID_5
	  3 arg: I2C CLK. should be 100KHZ, or 400KHz
	  4 arg: Source clock, should be set @ 19.2 MHz for V1
	         and 50MHz for V2
	    or Higher Rev
	 */
	   if( soc_ver >= BOARD_SOC_VERSION2 ){
	    dev = qup_blsp_i2c_init(BLSP_ID_2, QUP_ID_4, 100000, 50000000);
	  }
	  else{
	    dev = qup_blsp_i2c_init(BLSP_ID_2, QUP_ID_4, 100000, 19200000);
	  }
	if(!dev){
		printf("Failed to initializing\n");
		return;
	}

	//Received valid ptr
	printf("i2c_dev Ptr %p \n", dev);

	//Test Transfer
	msg.addr = 0x52;
	msg.flags = I2C_M_RD;
	msg.len = 10;
	msg.buf = buf;

	ret = qup_i2c_xfer(dev, &msg, 1);

	printf("qup_i2c_xfer returned %d \n", ret);

	for(i = 0; i < 10; i++)
		printf("%x ", buf[i]);

	printf("\n");
}
```

Output

```
i2c_dev Ptr 0xf950cbc
[64420] QUP IN:bl:8, ff:32, OUT:bl:8, ff:32
[64420] Polling Status for state:0x0
[64430] Polling Status for state:0x10
[64430] Polling Status for state:0x0
[64430] Polling Status for state:0x1
[64440] Polling Status for state:0x0
[64440] Polling Status for state:0x3
[64440] RD:Wrote 0x40a01a5 to out_ff:0xf9967110
[64450] Polling Status for state:0x0
[64450] Polling Status for state:0x1
[64450] idx:4, rem:1, num:1, mode:0
qup_i2c_xfer returned 1
ff ff ff ff ff ff ff ff ff ff
```

4.2.3 Debug LK I2C

This section provides debugging tips for situations where I2C fails for simple read/write operations.

1. Check SDA/SCL idling. Scope the bus to ensure that the SDA/SCL is idling at the high logic level. If it is not idling high, either there is a hardware configuration problem or the GPIO settings are invalid.
2. Set a breakpoint. Set a breakpoint after the qup_blsp_i2c_init function.

```
void my_i2c_test()
{
         if( soc_ver >= BOARD_SOC_VERSION2 ){
         dev = qup_blsp_i2c_init(BLSP_ID_2, QUP_ID_4, 100000,
          50000000);
         }
        else{
          dev = qup_blsp_i2c_init(BLSP_ID_2, QUP_ID_4, 100000,
           19200000);
        }
      if(!dev){ //Put break point @ this line.
            printf("Failed to initializing\n");
            return;
      }
```

3. Check the clock status. Check the QUP core clock and ensure that the BLSP_AHB clock is on by running testclock.cmm to dump all of the clock settings. This file is located in the following folders:

Chipset	Folder path
MSM	rpm_proc/core/systemdrivers/clock/scripts/<*chipset*>/testclock.cmm
APQ	boot_image/core/systemdrivers/clock/scripts/<*chipset*>/testclock.cmm
MDM	Contact QTI for support

4. Check GPIO configuration. Check the GPIO configuration register, GPIO_CFGn, to ensure that the GPIO settings are valid.

```
Physical Address: 0xFD511000 + (0x10 * n) = GPIO_CFGn
  n = GPIO #
  Example Address:
    0xFD511000 = GPIO_CFG0
    0xFD511010 = GPIO_CFG1

Bit definition for GPIO_CFGn
  Bits 31:11  Reserved
  Bit  10     GPIO_HIHYS_EN   Control the hihys_EN for GPIO
  Bit  9      GPIO_OE         Controls the Output Enable for GPIO
                              when in GPIO mode.

  Bits 8:6    DRV_STRENGTH    Control Drive Strength
                              000:2mA  001:4mA  010:6mA  011:8mA
                              100:10mA 101:12mA 110:14mA 111:16mA
  Bits 5:2    FUNC_SEL        Make sure Function is BLSP
                              Check Device Pinout for Correct Function
  Bits 1:0    GPIO_PULL       Internal Pull Configuration
                              00:No Pull   01: Pull Down
                              10:Keeper    11: Pull Up
```

NOTE: For I2C, QTI recommends 2 mA with no pull.

4.3 Configure kernel I2C

4.3.1 Code changes

The following files are used to configure a QUP core as an I2C in the kernel:

File type	Description
Device tree source	For MSM and APQ products: /kernel/arch/arm/boot/dts/qcom/<*chipset*>.dtsi Where <`chipset`> corresponds to the applicable chipset, for example: /kernel/arch/arm/boot/dts/qcom/msm8994.dtsi
Clock table	The clock nodes need to be added to the DTSI file. Project_Root/drivers/clk/qcom/clock-gcc-<chipset>.c
Pinctrl settings	The pin control table is located in the following file: /kernel/arch/arm/boot/dts/qcom/<*chipset*>-pinctrl.dtsi

I2C driver `i2c-msm-v2.c` supports BAM mode along with FIFO mode. Hence, it supports I2C Fast mode plus (up to 1 MHz).

The following procedure describes the steps required to configure and use any of the QUP cores (specifically, BLSP1_QUP1) as an I2C device for MSM8994 (for other platform, it is similar).

1. Create a device tree node. Modify the following file to add a new device tree node:

   1.a Old version of i2c driver that's based on qcom sps bam API (keyword : msm_i2c_bam_xfer in kernel/drivers/i2c/busses/i2c-msm-v2.c)

```
/kernel/arch/arm/boot/dts/qcom/msm8994.dtsi

/* If multiple I2Cs are registered, add aliases to
   identify the I2C Device ID.*/
aliases {
  i2c2 = &i2c_2; //i2c2 will be registered as /dev/i2c-2
};
i2c_2: i2c@f9924000 { //QUP_BASE Address for BLSP1_QUP1
       compatible = "qcom,i2c-msm-v2";
       #address-cells = <1>;
       #size-cells = <0>;
       reg-names = "qup_phys_addr", "bam_phys_addr";
       reg = <0xf9924000 0x1000>,
             <0xF9904000 0x19000>;

        /* Add BAM IRQ
           BLSP1: 238
           BLSP2: 239
       */
       interrupt-names = "qup_irq", "bam_irq"
       interrupts = <0 96 0>, <0 238 0>;

       /* Add BAM pipes, refer Table 4-x */
       qcom,bam-pipe-idx-cons = <14>;
       qcom,bam-pipe-idx-prod = <15>;

       qcom,clk-freq-out = <100000>;  //Output clock frequency
                                      //(can be upto 100KHz, or 400Khz)
       qcom,clk-freq-in  = <19200000>; //Source clock frequency
                                       //must be 19.2 MHz

       /* Select nodes for AHB and Core clocks
      In clock regime QUP clocks are labeled #1 to #6.
       So BLSP1 QUP1 is clk_gcc_blsp1_qup2_i2c_apps_clk   */
       clock-names = "iface_clk", "core_clk";
       clocks = <&clock_gcc clk_gcc_blsp1_ahb_clk>,
                <&clock_gcc clk_gcc_blsp1_qup2_i2c_apps_clk>;

       /* Pinctrl settings */
       pinctrl-names = "i2c_active", "i2c_sleep"; //Keep same names
       pinctrl-0 = <&i2c_2_active>;
       pinctrl-1 = <&i2c_2_sleep>;

       /* Noise rejection levels could set for SDA and SCL lines
          Accepted levels are 0-3 */
       qcom,noise-rjct-scl = <0>;
       qcom,noise-rjct-sda = <0>
       qcom,master-id = <86>; //to enable Bus scaling
};
```

1.b New version of i2c driver that makes use of the standard linux DMA framework (keyword : msm_i2c_dma_xfer in kernel/drivers/i2c/busses/i2c-msm-v2.c)

/kernel/arch/arm/boot/dts/qcom/msm8994.dtsi

```
/* If multiple I2Cs are registered, add aliases to
   identify the I2C Device ID.*/
aliases {
  i2c2 = &i2c_2; //i2c2 will be registered as /dev/i2c-2
};
i2c_2: i2c@f9924000 { //QUP_BASE Address for BLSP1_QUP1
       compatible = "qcom,i2c-msm-v2";
       #address-cells = <1>;
       #size-cells = <0>;
       reg-names = "qup_phys_addr";
       reg = <0xf9924000 0x1000>;
       interrupt-names = "qup_irq";
       interrupts = <0 96 0>;
       dmas = <&dma_blsp1 14 32 0x20000020 0x20>,
              <&dma_blsp1 15 64 0x20000020 0x20>;
       dma-names = "tx", "rx";


       qcom,clk-freq-out = <100000>;  //Output clock frequency
                                      //(can be upto 100KHz, or 400Khz)
       qcom,clk-freq-in  = <19200000>; //Source clock frequency
                                       //must be 19.2 MHz

       /* Select nodes for AHB and Core clocks
      In clock regime QUP clocks are labeled #1 to #6.
       So BLSP1 QUP1 is clk_gcc_blsp1_qup2_i2c_apps_clk   */
       clock-names = "iface_clk", "core_clk";
       clocks = <&clock_gcc clk_gcc_blsp1_ahb_clk>,
                <&clock_gcc clk_gcc_blsp1_qup2_i2c_apps_clk>;

       /* Pinctrl settings */
       pinctrl-names = "i2c_active", "i2c_sleep"; //Keep same names
       pinctrl-0 = <&i2c_2_active>;
       pinctrl-1 = <&i2c_2_sleep>;

       /* Noise rejection levels could set for SDA and SCL lines
          Accepted levels are 0-3 */
       qcom,noise-rjct-scl = <0>;
       qcom,noise-rjct-sda = <0>
       qcom,master-id = <86>; //to enable Bus scaling
};
```

/* If dma_blsp1 and dma_blsp2 are not defined in the device tree(msm8994.dtsi), please add it. */

```
dma_blsp1: qcom,sps-dma@f9904000 { /* BLSP1 */
        #dma-cells = <4>;
        compatible = "qcom,sps-dma";
        reg = <0xf9904000 0x19000>;
        interrupts = <0 238 0>;
        qcom,summing-threshold = <10>;
};

dma_blsp2: qcom,sps-dma@f9944000 { /* BLSP2 */
        #dma-cells = <4>;
        compatible = "qcom,sps-dma";
        reg = <0xf9944000 0x19000>;
        interrupts = <0 239 0>;
        qcom,summing-threshold = <10>;
};
```

For details, refer to `/kernel/Documentation/devicetree/bindings/i2c/i2c-msm-v2.txt`.

2. Set the Pinctrl settings.
   a. Please refer to the chipset device specification to check BLSP gpio assignment, and refer to the chipset software interface to check BLSP gpio configuration.
   b. Open the `.dtsi` file located at:

      `kernel/arch/arm/boot/dts/qcom/<chipset>-pinctrl.dtsi`

   c. Modify the pin control settings as shown in the following example. For more information, refer to pin control documentation located at `/kernel/Documentation/devicetree/bindings/pinctrl/msm-pinctrl.txt`.

```
&soc {
  tlmm_pinmux: pinctrl@fd510000 {

//snip

 i2c2_active {
   qcom,pins = <&gp 6>, <&gp 7>;
   qcom,num-grp-pins = <2>;
   qcom,pin-func = <3>;  // OEM must get the right gpio func number from the
software interface: TLMM chapter!
   label = "i2c2_active";
   i2c_2_active: default {
     drive-strength = <2>;
     bias-disable;
   };
 };

 i2c2_sleep {
   qcom,pins = <&gp 6>, <&gp 7>;
   qcom,num-grp-pins = <2>;
   qcom,pin-func = <0>;
   label = "i2c2_sleep";
   i2c_2_sleep: sleep` {
     drive-strength = <2>;
     bias-disable;
   };
 };
```

From kernel 3.14 onwards that MSM8996 is using, pinctrl framework changes a bit. Refer to the following demo to configure the I2C GPIOs:

```
&soc {
        tlmm: pinctrl@01010000 {
//snip

i2c_2 {
 i2c_2_active: i2c_2_active {
   i2c_2 {
          pins = “gpio6”,”gpio7”;
          function = “blsp_i2c8”;  //For how to get the function name,
please check the pinctrl driver code: kernel/drivers/pinctrl/qcom/pinctrl-
msm8996.c
           drive-strength = <2>;
           bias-disable;
   };
 };

 i2c_2_sleep: i2c_2_sleep {
   i2c_2 {
          pins = “gpio6”,”gpio7”;
          functions = “blsp_i2c8”; //For how to get the function name of
each pin, please check the pinctrl driver code:
kernel/drivers/pinctrl/qcom/pinctrl-msm8996.c
          drive-strength = <2>;
          bias-pull-up;
   };
 };
};
```

3. Configure BAM pipes for the SPI master by modifying the TrustZone source code. See section *UART→Configure kernel high-speed UART(3.4.1)→Code changes*, step 3.

4. Verify the I2C bus. Ensure that the bus is registered. If you entered all of the information properly, you should see I2C bus registered under /dev/i2c-#, where cell-index matches the bus number.

```
adb shell --> Get adb shell
cd /dev/
ls i2c* --> to List all the I2C buses
root@android:/dev # ls i2c*
ls i2c*
i2c-0
i2c-10
i2c-2
```

5. Test the bus. If the bus is properly registered, test it using a simple test program. The test programs shown in the following example is located in the following directory:

```
/vendor/qcom/proprietary/kernel-tests/i2c-test/
```

Make file – Android.mk

```
ifeq ($(TARGET_ARCH),arm)

LOCAL_PATH := $(call my-dir)
commonSources :=

include $(CLEAR_VARS)
LOCAL_MODULE := i2c-test
LOCAL_SRC_FILES += $(commonSources) i2c-test.c
LOCAL_C_INCLUDES := $(TARGET_OUT_INTERMEDIATES)/KERNEL_OBJ/usr/include/
LOCAL_ADDITIONAL_DEPENDENCIES :=
$(TARGET_OUT_INTERMEDIATES)/KERNEL_OBJ/usr
LOCAL_MODULE_TAGS := optional debug
LOCAL_MODULE_PATH := $(TARGET_OUT_DATA)/kernel-tests

LOCAL_SHARED_LIBRARIES := \
        libc \
        libcutils \
        libutils

include $(BUILD_EXECUTABLE)

endif
```

4.3.2 Disable BAM mode

To disable BAM mode for transfers greater than FIFO size = 64 bytes, you could do either of the following:

- Set the following field in DTS:

```
qcom,bam-disable;
```

- Run the following ADB shell command (I2C master driver debugfs feature must be enabled to access to debugfs node, CONFIG_I2C_MSM_PROF_DBG=y):

```
echo 1 > /sys/kernel/debug/<device_address>.i2c/xfer-force-mode //When
xfer-force-mode is set, block mode will be used during transfer instead
of BAM and FIFO mode.
```

For example, for BLSP1_QUP1 on MSM8994,

```
echo 1 > /sys/kernel/debug/f9924000.i2c/xfer-force-mode
```

After BAM mode is disabled, I2C master driver automatically downgrades to Block mode that triggers more interrupts during I2C transfer and decreases CPU performance. Hence, BAM disable operation is not suggested for normal usecase instead of test purpose.

4.3.3 Test I2C master

QTI provides a demo code for I2C master test from user space at:

vendor/qcom/proprietary/kernel-tests/i2c-msm/

In engineer build, the binary is located at /data/kernel-tests/i2c-msm-test.

The test procedure is as follows:

1. Find I2C master :

```
root@msm8994:/sys/kernel/debug/f9928000.i2c # ls /dev/i2c*
ls /dev/i2c*
/dev/i2c-2
/dev/i2c-5
/dev/i2c-6
```

2. Confirm/change file attribute of i2c-msm-test to ensure that it has execute access:

```
cd /data/kernel-tests
root@msm8994:/data/kernel-tests # chmod 777 i2c-msm-test
chmod 777 i2c-msm-test
root@msm8994:/data/kernel-tests # ls -l i2c-msm-test
ls -l i2c-msm-test
-rwxrwxrwx root     root        22712 2015-06-12 07:07 i2c-msm-test
root@msm8994:/data/kernel-tests #
```

3. Run i2c-msm-test to test I2C master:

```
root@msm8994:/data/kernel-tests # ./i2c-msm-test -D /dev/i2c-6 -s
0x52 -p
./i2c-msm-test -D /dev/i2c-6 -s 0x52 -p
Error #684 Could not read from EEPROM at address 0x52
 err#:107:Transport endpoint is not connected
root@msm8994:/data/kernel-tests #

Because there is no i2c slave device with address 0x52, kernel log
shows a NACK error:
<3>[ 2984.270645] i2c-msm-v2 f9928000.i2c: NACK: slave not
responding, ensure its powered, Invalid slave addr, I2C transfer
failed, : msgs(n:2 cur:0 rx) bc(rx:1 tx:2) mode:FIFO slv_addr:0x52
MSTR_STS:0x0d1300c8 OPER:0x00008010
```

At this point, the debugging should focus on the slave device to ensure that it is properly powered up and ready to accept I2C transfer.

The following error message may be due to multiple issues:

- Invalid software configuration
- Invalid hardware configuration
- Slave device issues

For more information about i2c-msm-test, go through the source code at vendor/qcom/proprietary/kernel-tests/i2c-msm/i2c-msm-test.c

and run:

./i2c-msm-test --help.

4.3.4 Register a slave device using the device tree

After the I2C bus is properly verified, you can create a slave device driver and register it with I2C bus. See the following files for examples:

- For an I2C slave device, refer to `msm8994-cdp.dtsi`.
- For Atmel Touch Screen driver registration, refer to `atmel_mxt_ts.c`.

The following examples show the minimum requirement for properly registering a slave device using the device tree.

1. Create a device tree node. Open the following file and add a device tree node:

```
/kernel/arch/arm/boot/dts/<chipset>-cdp.dtsi


i2c@f9924000 { //I2C BUS Slave connected
  qcom_i2c_test@52 { //Slave driver and slave Address
          compatible ="qcom,i2c-test";//Manufacture, model
   reg = <0x52>; //Slave Address
   /*
   * Following elements are optional.  For more detail
   * settings please check atmel touch screen driver
   * or sdcc device tree.
   */
   interrupt-parent = <&msmgpio>; //GPIO handler
   interrupts = <61 0x2>; //GPIO # will be converted to gpio_irq #
   qcom_i2c_test,irq-gpio = <&msmgpio 61 0x00>; //Pass a GPIO
   };
};
```

2. Create or modify the slave driver. The following provides an example of the slave driver.

NOTE: `i2c_transfer()` is a nonblocking call. The buffer passed by a client is freed when the function exits, while it still might be needed on the master side for BAM transfer. Hence, the client should allocate buffers from Heap.

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/delay.h>
#include <linux/i2c.h>
#include <linux/interrupt.h>
#include <linux/slab.h>
#include <linux/gpio.h>
#include <linux/debugfs.h>
#include <linux/seq_file.h>
#include <linux/regulator/consumer.h>
#include <linux/string.h>
#include <linux/of_gpio.h>

#ifdef CONFIG_OF //Open firmware must be defined for dts usage
static struct of_device_id qcom_i2c_test_table[] = {
      { .compatible = "qcom,i2c-test",}, //Compatible node must
                                         //match dts
      { },
};
#else
#define qcom_i2c_test_table NULL
#endif

//I2C slave id supported by driver
static const struct i2c_device_id qcom_id[] = {
      { "qcom_i2c_test", 0 },
      { }
};

static int i2c_test_test_transfer(struct i2c_client *client)
{
  struct i2c_msg xfer; //I2C transfer structure
  u8 *buf = kmalloc(1, GFP_ATOMIC); //allocate buffer from Heap since
i2c_transfer() is non-blocking call
  buf[0] = 0x55;  //data to transfer
  xfer.addr = client->addr;
  xfer.flags = 0;
  xfer.len = 1;
  xfer.buf = buf;

  return i2c_transfer(client->adapter, &xfer, 1);
}

static int i2c_test_probe(struct i2c_client *client,
            const struct i2c_device_id *id)
{
  int irq_gpio = -1;
  int irq;
  int addr;
  //Parse data using dt.
  if(client->dev.of_node){
    irq_gpio = of_get_named_gpio_flags(client->dev.of_node,
"qcom_i2c_test,irq-gpio", 0, NULL);
  }
  irq = client->irq; //GPIO irq #. already converted to gpio_to_irq
  addr = client->addr; //Slave Addr
  dev_err(&client->dev, "gpio [%d] irq [%d] gpio_irq [%d] Slaveaddr
[%x] \n", irq_gpio, irq,
    gpio_to_irq(irq_gpio), addr);
```

```
    //You can initiate a I2C transfer anytime
    //using i2c_client *client structure
    i2c_test_test_transfer(client);

    return 0;
}

//I2C Driver Info
static struct i2c_driver i2c_test_driver = {
      .driver = {
            .name = "qcom_i2c_test",
            .owner= THIS_MODULE,
            .of_match_table = qcom_i2c_test_table,
      },
      .probe      = i2c_test_probe,
      .id_table       = qcom_id,
};
```

In the kernel log, the following message indicates the device tree was successfully configured.

```
<3>[    2.670731] qcom_i2c_test 2-0052: gpio [61] irq [306] gpio_irq [306]
                  Slaveaddr [52]
```

4.3.5 Noise rejection on I2C lines

Sometimes there could be noise seen on I2C lines due to other signal interference. I2C hardware allows us to set the sampling level (0–3) to reject short low pulses. It specifies how many TCXO cycles of logic low on SDA/SCL would be considered as valid logic low.

- 0x0 – legacy mode
- 0x01 – one cycle wide low pulse is rejected
- 0x2 – two cycles wide low pulse is rejected
- 0x3 – three cycles wide low pulse is rejected

These values could be set in DTS using following fields:

```
qcom,noise-rjct-scl = <1>;
qcom,noise-rjct-sda = <1>;
```

By default, these values are 0.

4.3.6 Setting I2C clock dividers

The I2C specification has set limits on the HIGH and LOW period of the I2C clock pulse.

Symbol	Parameter	Conditions	Standard-mode		Fast-mode		Fast-mode Plus		Unit
			Min	Max	Min	Max	Min	Max	
f_{SCL}	SCL clock frequency		0	100	0	400	0	1000	kHz
t_{LOW}	LOW period of the SCL clock		4.7	-	1.3	-	0.5	-	µs
t_{HIGH}	HIGH period of the SCL clock		4.0	-	0.6	-	0.26	-	µs

To meet these limits, QUP register I2C_CLK_CTL can be programmed for setting I2C clock dividers.

Bits	Name	Description
23:16	HIGH_TIME_DIVIDER_VALUE	Allows setting SCL duty cycle to non 50%. If this value is zero than legacy mode is used. If this value is non-zero than it will be used as the SCL high time counter and FS_DIVIDER_VALUE will be used as the low time counter. Minimum value is 0x7.
7:0	FS_DIVIDER_VALUE	The value in this register represents the clock period multiplier in fast/standard (FS) mode. Minimum value is 0x7. When HIGH_TIME_DIVIDER_VALUE=0: I2C_FS_CLK = I2C_CLK/(2*(FS_DIVIDER_VALUE+3)) When HIGH_TIME_DIVIDER_VALUE!=0: I2C_FS_CLK = I2C_CLK/(FS_DIVIDER_VALUE+HIGH_TIME_DIVIDER_VALUE+6)

4.3.6.1 Default values

How to calculate the default value of fs_div and hs_div if hs_div is not 0:

I2C_FS_CLK = I2C_CLK/(fs_div+hs_div+6)

100K transfer rate as example:
19.2M (CXO) /100K(transfer rate) = fs_div+hs_div+6
fs_div+hs_div = 186
Generally, fs_div = 2hs_div (For detailed information refer to 4.3.6.3)
That's why default is: 124/62 under 100K transfer rate.

Output clock frequency	FS divider	HS divider
100 kHz	124	62
400 kHz	28	14
1 MHz	8	5
Other non-standard frequency	OEM must calculate the FS and HS divider using the formula in Section 4.3.6.1.	

4.3.6.2 Set values

Clock divider values can vary across different boards to meet the I2C specification limits. You could override the default values set in the driver by using the following DTS fields:

```
i2c_2: i2c@f9924000 { /* BLSP1 QUP1 */
   //snip
    qcom,fs-clk-div = <28>;
    qcom,high-time-clk-div = <14>;
};
```

The FS divider value is responsible for the LOW period (Tlow). Reducing it by 1 shortens Tlow by 52 ns (assuming the source clock is TCXO 19.2 MHz).

4.3.6.3 Dividers vs clock frequency

The SCL period is calculated as:

```
T = TCXO * ( (FS_DIV+HT_DIV) + 6 + NR)  + Trise
   where
   TCXO is 52ns
   NR is Noise Rejection level
   Trise is SCL Rise time
```

Trise will be > 0, hence output clock (1/T) will be lesser than what we set, for example, 400 kHz. This is shown in Figure 4-1 and Figure 4-2.

Figure 4-1 Output clock is less than 400 kHz due to added rise time

Figure 4-2 Output clock is 400 kHz due to excluded rise time

The divider ratio, FS_DIV/HTD, should be 2:1. Tweak the divider values to maintain this ratio and get a lesser sum so that a higher output clock can be generated.

Example:

100K transfer rate as example:
19.2M (CXO) /100K(transfer rate) = fs_div+hs_div+6
fs_div+hs_div = 186
The default fs_div/hs_div is: 124/62 under 100K transfer rate.

If OEMs want to achieve a longer output clock high time, they can change the hs_div to 63 or a larger value, so that fs_div should be (186 – hs_div) = 123, and vice versa, they can change the hs_div to a smaller value.

4.4 I2C power management

I2C slave devices must register system suspend/resume (SYSTEM_PM_OPS) handlers with the power management framework to ensure that no I2C transactions are initiated after the I2C master is suspended.

Example

```
/* Register PM Hooks */
static const struct dev_pm_ops i2c_test_pm_ops = {
       SET_SYSTEM_SLEEP_PM_OPS(
                i2c_test_suspend, //Get call when suspend is happening
                i2c_test_resume //Get call when resume is happening
        )
};

//I2C Driver Info
static struct i2c_driver i2c_test_driver = {
      .driver = {
              .pm = &i2c_test_pm_ops,
      },
      .probe          = i2c_test_probe,
      .id_table       = qcom_id,
};

/* System Going to Suspend*/
static int i2c_test_suspend(struct device *device)
{
 /*
   * Properly set slave device to suspend (I2C transactions are OK)
   * Set a suspend flag
   * No more I2C transaction should occur until i2c_test_resume is called
 */
      return 0;
}

static int i2c_test_resume(struct device *device)
{
  /*
    * Remove slave device from suspend (I2C transactions are OK)
    * Clear suspend flag
  */
      return 0;
}
```

4.5 I2C transfer pseudocode

The following example shows an I2C transfer for a typical read register.

```
u8 buf[2]
u8 val[2]
struct i2c_msg xfer[2]

/* Reading data from a 16 bit addressing device */
buf[0] = reg 0xff; //lower bits
buf[1] = (reg >> 8)      0xff; //upper bits

/* Program register to read */
xfer[0].addr = client->addr;
xfer[0].flags = 0;
xfer[0].len = 2;
xfer[0].buf = buf; //16 bit reg

/* Read data */
xfer[1].addr = client->addr;
xfer[1].flags = I2C_M_RD;
xfer[1].len = len;
xfer[1].buf = val;

/* Perform the transfer */
i2c_transfer(client->adapter, xfer, 2);
```

The following code explains the Do the Transfer part.

```
func: set_read_mode(){
 * if read length < FIFO_SIZE set QUP_MX_READ_COUNT=read length
 * if read length > FIFO_SIZE set:
   QUP_MX_INPUT_COUNT = read length
   QUP_IO_MODE |= INPUT_BLOCK_MODE
}
func: set_write_mode(){
 * Calculate the total length of transfer.  If next message is a write
   and slave address same then combine to total transfer
 * Configure QUP_IO_MODES=PACK_EN|UNPACK\_EN
 * if total length >= FIFO_SIZE enable Write BLOCK MODE QUP_IO_MODES
 * Check if any read messages for slave address, if so call
   func:set_read_mode
 * if using block mode program QUP_MX_OUTPUT_COUNT = total length
}

...

func: isr handler{
 * Read QUP_I2C_MASTER_STATUS
 * Read QUP_ERROR_FLAGS
 * Read QUP_OPERATIONALS
 * Check for any Error, if Error, clear Error status
```

```
   and reset QUP controller and return
 * Any output service done, clear it.
 * if input service done, clear the status.
 * Issue complete done signal
}
...

Enter:
if (doing a read transfer) {
 call func:set_read_mode()
}
else{
 call func:set_write_mode()
}
* Change QUP to Run State
* Program I2C_MASTER_CLK_CTL register
* Change QUP to PAUSE state
* Program Output FIFO
* TAG_START|address
* TAG_OUTPUT_DATA | data
* Increment to next message
* Program Output FIFO
* TAG_START|address
* TAG_OUT_REC | # of bytes
* Change to Run State
* Wait for completion signal
--Should receive interrupt--
--and Completion signal
* Read the input buffer and copy the data
* if any more msg left go to "Enter"
  else disable irq, update pm_last_busy
* return # of msg processed
```

4.5.1 QUP operational states

The QUP sub-block maintains three operational states:

- RESET_STATE (00) – The default state after a software or hardware reset of the QUP core. The mini-core and FIFOs are held in reset.
- RUN_STATE (01) – The mini-core is brought out of reset, and the protocol-related activity is initiated based on the register states.
- PAUSE_STATE (11) – The mini-core stops initiating new transfers. FIFOs can be filled during this stage.

4.5.2 I2C V1 TAG

The I2C mini-core uses a tagging mechanism to transfer specific data to/from QUP FIFOs. A data word written to FIFO is composed of an 8-bit TAG, and an 8-bit value is associated with each TAG.

Table 4-8 I2C V1 TAG

TAG name	TAG value	DATA field	Comments
NOOP	0x00	0xCC	Wait (0xCC*9) number of I2C clock cycles
START	0x01	0xAA	0xAA – 7-bit slave address + read/write bit
MO_DATA	0x02	0xDD	0xDD – Master output data
MO_STOP	0x03	0xDD	0xDD – Master output data, output data with a STOP
MI_REC	0x04	0xCC	0xCC – Number of bytes to receive XX controller automatically generates a NACK and stop condition
MI_DATA	0x05	0xDD	0xDD – Master input data
MI_STOP	0x06	0xDD	0xDD – Last byte of master input
MI_NACK	0x07	0xFF	Invalid input data

4.6 Debug I2C driver

4.6.1 Debug checklist

This section provides debugging tips for situations where I2C fails for simple read/write operations.

1. Check SDA/SCL idling (logic high). Scope the bus to ensure that the SDA/SCL is idling at the high logic level. If it is not idling high, either there is a hardware configuration problem or the GPIO settings are invalid.

Example 1:

- [2.075055] i2c-msm-v2 78b6000.i2c: NACK: slave not responding, ensure its powered, Invalid slave addr, I2C transfer failed, : msgs(n:2 cur:0 rx) bc(rx:1 tx:1) mode:FIFO slv_addr:0x63 MSTR_STS:0x011363c8 OPER:0x00000010
- Refer to chipset software interface about the register: PERIPH_SS_BLSP*_BLSP_QUP*_I2C_MASTER_STATUS
- MSTR_STS (I2C Master Status Register)- 0x011363C8
- Bit 26&27 are 0, that means SDA and SCL lines are low.
- Bit 3 is 1, that means NACK.

Example 2:

- [01-01 00:12:38.276] i2c-msm-v2 78b6000.i2c: arb_lost, I2C transfer failed, : msgs(n:2 cur:0 tx) bc(rx:1 tx:1) mode:FIFO slv_addr:0x63 MSTR_STS:0x081121d0 OPER:0x00000e60slv_addr:0x63 MSTR_STS:0x011363c8 OPER:0x00000010
- Refer to chipset software interface about the register: PERIPH_SS_BLSP*_BLSP_QUP*_I2C_MASTER_STATUS
- MSTR_STS (I2C Master Status Register)- 0x081121d0

- Bit 26 is 0, that means SDA line low.
- Bit 4 is 1, that means ARB_LOST.

2. Set a breakpoint at the line where the error message is coming, for example, at the `Transaction timed out` message:

```
static int i2c_msm_qup_post_xfer(struct i2c_msm_ctrl *ctrl, int err)
{
…
        /* flush dma data and reset the qup core in timeout error.
         * for other error case, its handled by the ISR
         */
Drop a breakpoint here.
        if (ctrl->xfer.err & BIT(I2C_MSM_ERR_TIMEOUT)) {
….
}
```

3. Check the clock status. Check the QUP core clock and ensure that the BLSP_AHB clock is on by running testclock.cmm to dump all the clock settings. This script is located at:

```
rpm_proc/core/systemdrivers/clock/scripts/<chipset>/testclock.cmm
```

4. Check the GPIO configuration register, GPIO_CFGn, to ensure that the GPIO settings are valid.

```
Physical Address: 0xFD511000 + (0x10 * n) = GPIO_CFGn
  n = GPIO #
  Example Address:
    0xFD511000 = GPIO_CFG0
    0xFD511010 = GPIO_CFG1

Bit definition for GPIO_CFGn
  Bits 31:11  Reserved
  Bit  10     GPIO_HIHYS_EN    Control the hihys_EN for GPIO
  Bit  9      GPIO_OE          Controls the Output Enable for GPIO
                               when in GPIO mode.
  Bits 8:6    DRV_STRENGTH     Control Drive Strength
                               000:2mA  001:4mA  010:6mA  011:8mA
                               100:10mA 101:12mA 110:14mA 111:16mA
  Bits 5:2    FUNC_SEL         Make sure Function is BLSP
                               Check Device Pinout for Correct Function
  Bits 1:0    GPIO_PULL        Internal Pull Configuration
                               00:No Pull   01: Pull Down
                               10:Keeper    11: Pull Up
```

Note: For I2C, QTI recommends 8 mA with no pull.

5. To get more debug info, we can make use the following debug fs interface:

```
/sys/kernel/debug/*.i2c/
Example, f9924000.i2c:
adb root
adb shell
#cd /sys/kernel/debug/ f9924000.i2c
# ls
ls
bus-clear
dbg-lvl
dump-regs
xfer-force-mode
# echo 1 > dump-regs
#dmesg
<6>[  147.884567] i2c-msm-v2 f9924000.i2c: #4035 pm_runtime: resuming...
<6>[  147.884610] i2c-msm-v2 f9924000.i2c: #3951 resuming...
<3>[  147.887582] i2c-msm-v2 f9924000.i2c: QUP_CONFIG  :0x00000287 N:0x7
MINI_CO
RE:I2C NO_INPUT
<3>[  147.896572] i2c-msm-v2 f9924000.i2c: QUP_STATE   :0x0000001d
STATE:Run VAL
ID MAST_GEN
<3>[  147.903591] i2c-msm-v2 f9924000.i2c: QUP_IO_MDS  :0x0000c0a5
IN_BLK_SZ:16
IN_FF_SZ:x4 blk sz OUT_BLK_SZ:16 OUT_FF_SZ:x4 blk sz UNPACK PACK
<3>[  147.916519] i2c-msm-v2 f9924000.i2c: QUP_ERR_FLGS:0x00000000
<3>[  147.921934] i2c-msm-v2 f9924000.i2c: QUP_OP      :0x000004c0
OUT_FF_FUL IN
_FF_FUL MX_OUT_DN
<3>[  147.930875] i2c-msm-v2 f9924000.i2c: QUP_OP_MASK :0x00000000
<3>[  147.935868] i2c-msm-v2 f9924000.i2c: QUP_I2C_STAT:0x00112100 BUS_ACTV
CLK_
STATE:0x1 O_FSM_STAT:0x1 I_FSM_STAT:0x2
<3>[  147.946315] i2c-msm-v2 f9924000.i2c: QUP_MSTR_CLK:0x000e001c
FS_DIV:0x1c H
I_TM_DIV:0xe
<3>[  147.954234] i2c-msm-v2 f9924000.i2c: QUP_IN_DBG  :0x00000000
<3>[  147.959918] i2c-msm-v2 f9924000.i2c: QUP_OUT_DBG :0x00000000
<3>[  147.965632] i2c-msm-v2 f9924000.i2c: QUP_IN_CNT  :0x00000000
<3>[  147.971391] i2c-msm-v2 f9924000.i2c: QUP_OUT_CNT :0x00000000
<3>[  147.977091] i2c-msm-v2 f9924000.i2c: MX_RD_CNT   :0x00000000
<3>[  147.982850] i2c-msm-v2 f9924000.i2c: MX_WR_CNT   :0x00000006
<3>[  147.988548] i2c-msm-v2 f9924000.i2c: MX_IN_CNT   :0x00000000
<3>[  147.994372] i2c-msm-v2 f9924000.i2c: MX_OUT_CNT  :0x00000000
<6>[  149.040449] i2c-msm-v2 f9924000.i2c: #4023 pm_runtime: suspending...
<6>[  149.040491] i2c-msm-v2 f9924000.i2c: #3937 suspending...
```

4.6.2 Debug log

4.6.2.1 i2c-msm-v2.c – FIFO mode

Here is a sample log for a combined message of 1-byte write, 6-bytes read. To enable these logs, define the following macro in `i2c-msm-v2.c`:

```
#define DEBUG

// Transfer begins. FIFO mode used
//#1392 gives the Line number for print i.e Line 1392
<6>[   25.792522] i2c-msm-v2 f9924000.i2c: #1392 Starting FIFO transfer

//Programmed Registers for transfer
<6>[   25.798561] i2c-msm-v2 f9924000.i2c: QUP state after programming for
next transfers
<3>[   25.806169] i2c-msm-v2 f9924000.i2c: QUP_CONFIG  :0x00000207 N:0x7
MINI_CORE:I2C
<3>[   25.813652] i2c-msm-v2 f9924000.i2c: QUP_STATE   :0x0000001d
STATE:Run VALID MAST_GEN
<3>[   25.821552] i2c-msm-v2 f9924000.i2c: QUP_IO_MDS  :0x0000c0a5
IN_BLK_SZ:16 IN_FF_SZ:x4 blk sz OUT_BLK_SZ:16 OUT_FF_SZ:x4 blk sz UNPACK
PACK
<3>[   25.834048] i2c-msm-v2 f9924000.i2c: QUP_ERR_FLGS:0x00000000
<3>[   25.839776] i2c-msm-v2 f9924000.i2c: QUP_OP      :0x00000000
<3>[   25.845488] i2c-msm-v2 f9924000.i2c: QUP_OP_MASK :0x00000000
<3>[   25.851239] i2c-msm-v2 f9924000.i2c: QUP_I2C_STAT:0x0c110000
O_FSM_STAT:0x1 I_FSM_STAT:0x2 SDA SCL
<3>[   25.860264] i2c-msm-v2 f9924000.i2c: QUP_MSTR_CLK:0x000e001c
FS_DIV:0x1c HI_TM_DIV:0xe
<3>[   25.868232] i2c-msm-v2 f9924000.i2c: QUP_IN_DBG  :0x00000000
<3>[   25.874014] i2c-msm-v2 f9924000.i2c: QUP_OUT_DBG :0x00000000
<3>[   25.879743] i2c-msm-v2 f9924000.i2c: QUP_IN_CNT  :0x00000000
<3>[   25.885420] i2c-msm-v2 f9924000.i2c: QUP_OUT_CNT :0x00000000
<3>[   25.891171] i2c-msm-v2 f9924000.i2c: MX_RD_CNT   :0x00000008
<3>[   25.896876] i2c-msm-v2 f9924000.i2c: MX_WR_CNT   :0x00000009
<3>[   25.902625] i2c-msm-v2 f9924000.i2c: MX_IN_CNT   :0x00000000
<3>[   25.908336] i2c-msm-v2 f9924000.i2c: MX_OUT_CNT  :0x00000000

//First message is 1-byte Write.  So tags used are START, DATAWRITE
<6>[   25.914090] i2c-msm-v2 f9924000.i2c: tag.val:0x1824081 tag.len:4
val:0x01824081 START:0x40 DATAWRITE:1
<6>[   25.923370] i2c-msm-v2 f9924000.i2c: #1163 OUT-FIFO:0x01824081
<6>[   25.929721] i2c-msm-v2 f9924000.i2c: data: 0xe3 0xbc 0xbf 0xce

//Second message is 6-byte Read and its the last message.  So tags used are
START, DATARD_STOP
```

```
<6>[   25.935075] i2c-msm-v2 f9924000.i2c: tag.val:0x6874181 tag.len:4
val:0x06874181 START:0x41 DATARD_and_STOP:6
<6>[   25.944906] i2c-msm-v2 f9924000.i2c: #1163 OUT-FIFO:0x874181e3
<6>[   25.950716] i2c-msm-v2 f9924000.i2c: #1163 OUT-FIFO:0x00000006

//Slave address is 0x20. Total messages in the transfer are 2.
// From here onwards, we would track time taken for the transfer.
Currently, 0.000 ms in the transfer
<6>[   25.998372] i2c-msm-v2 f9924000.i2c: -->.000ms XFER_BEG msg_cnt:2
addr:0x20

//First message is Write for 1 byte
<6>[   26.005299] i2c-msm-v2 f9924000.i2c:   0.000ms XFER_BUF msg[0] pos:0
adr:0x20 len:1 is_rx:0x0 last:0x0

//Second message is Read for 6 bytes, and is the last one in the transfer
<6>[   26.014605] i2c-msm-v2 f9924000.i2c:   0.001ms XFER_BUF msg[1] pos:0
adr:0x20 len:6 is_rx:0x1 last:0x1

//Received QUP IRQ(96+32 = 128), ISR called
<6>[   26.088820] i2c-msm-v2 f9924000.i2c: 164.089ms  IRQ_BEG irq:128
<6>[   26.094708] i2c-msm-v2 f9924000.i2c: 176.233ms  IRQ_END
MSTR_STTS:0x8345b00 QUP_OPER:0x140 ERR_FLGS:0x0
<6>[   26.104101] i2c-msm-v2 f9924000.i2c:             |-> QUP_OPER:0x140
OUT_FF_FUL OUT_SRV_FLG

//Transfer complete successfully.
//Total time taken=205.850ms
<6>[   26.138824] i2c-msm-v2 f9924000.i2c: 205.850ms XFER_END ret:2
err:[NONE] msgs_sent:2 BC:17 B/sec:82 i2c-stts:OK
```

4.6.2.2 i2c-msm-v2.c – BAM mode

```
// Transfer begins. BAM mode used
//#2363 gives the Line number for print i.e Line 2363
<6>[   29.938056] i2c-msm-v2 f9924000.i2c: #2363 Starting BAM transfer

//Address for driver's bookkeeping BAM structure
<6>[   29.944060] i2c-msm-v2 f9924000.i2c: #2289 initializing
BAM@0xffffffc0cebf0000

//is_init gets set to TRUE at the end of init API
<6>[   29.952219] i2c-msm-v2 f9924000.i2c: #2114 Calling BAM producer pipe
init. is_init:0
<6>[   29.968194] i2c-msm-v2 f9924000.i2c: #2114 Calling BAM consumer pipe
init. is_init:0

//BAM pipe addresses
```

```
<6>[   29.976244] i2c-msm-v2 f9924000.i2c: #1849 vrtl:0xffffff80017ef010
phy:0xdb4af010 val:0x1824081 sizeof(dma_addr_t):8
<6>[   29.986373] i2c-msm-v2 f9924000.i2c: #1849 vrtl:0xffffff80017ef018
phy:0xdb4af018 val:0x50874181 sizeof(dma_addr_t):8

//Programmed Registers for transfer
<3>[   30.004550] i2c-msm-v2 f9924000.i2c: QUP_CONFIG  :0x00000207 N:0x7
MINI_CORE:I2C
<3>[   30.012015] i2c-msm-v2 f9924000.i2c: QUP_STATE   :0x0000001d
STATE:Run VALID MAST_GEN
<3>[   30.019903] i2c-msm-v2 f9924000.i2c: QUP_IO_MDS  :0x0000fca5
IN_BLK_SZ:16 IN_FF_SZ:x4 blk sz OUT_BLK_SZ:16 OUT_FF_SZ:x4 blk sz UNPACK
PACK INP_MOD:BAM OUT_MOD:BAM
<3>[   30.034494] i2c-msm-v2 f9924000.i2c: QUP_ERR_FLGS:0x00000000
<3>[   30.040207] i2c-msm-v2 f9924000.i2c: QUP_OP      :0x00000000
<3>[   30.045954] i2c-msm-v2 f9924000.i2c: QUP_OP_MASK :0x00000300
OUT_SRVC_MASK IN_SRVC_MASK
<3>[   30.054029] i2c-msm-v2 f9924000.i2c: QUP_I2C_STAT:0x0c110000
O_FSM_STAT:0x1 I_FSM_STAT:0x2 SDA SCL
<3>[   30.063055] i2c-msm-v2 f9924000.i2c: QUP_MSTR_CLK:0x000e001c
FS_DIV:0x1c HI_TM_DIV:0xe
<3>[   30.071023] i2c-msm-v2 f9924000.i2c: QUP_IN_DBG  :0x00000000
<3>[   30.076768] i2c-msm-v2 f9924000.i2c: QUP_OUT_DBG :0x00000000
<3>[   30.082496] i2c-msm-v2 f9924000.i2c: QUP_IN_CNT  :0x00000000
<3>[   30.088210] i2c-msm-v2 f9924000.i2c: QUP_OUT_CNT :0x00000000
<3>[   30.093955] i2c-msm-v2 f9924000.i2c: MX_RD_CNT   :0x00000000
<3>[   30.099669] i2c-msm-v2 f9924000.i2c: MX_WR_CNT   :0x00000000
<3>[   30.105413] i2c-msm-v2 f9924000.i2c: MX_IN_CNT   :0x00000000
<3>[   30.111127] i2c-msm-v2 f9924000.i2c: MX_OUT_CNT  :0x00000000
<6>[   30.116872] i2c-msm-v2 f9924000.i2c: #1934 Going to enqueue 2 buffers
in BAM

//First message is 1-byte Write. So tags used are START, DATAWRITE
<6>[   30.123906] i2c-msm-v2 f9924000.i2c: #1955 queueing bam tag
val:0x01824081 START:0x40 DATAWRITE:1
<6>[   30.132773] i2c-msm-v2 f9924000.i2c: #1984 Queue data buf to consumer
pipe desc(phy:0xcbc2fcc0 len:1) EOT:0 NWD:0

//Second message is 80-bytes Read, and is the last one. Tags used are
START, DATARD_and_STOP
<6>[   30.143005] i2c-msm-v2 f9924000.i2c: #1955 queueing bam tag
val:0x50874181 START:0x41 DATARD_and_STOP:80
<6>[   30.152465] i2c-msm-v2 f9924000.i2c: #1901 queuing input tag buf
len:2 to prod

//Slave address is 0x20. Total messages in the transfer are 2.
// From here onwards, we would track time taken for the transfer.
Currently, 0.000 ms in the transfer
```

```
<6>[   30.219029] i2c-msm-v2 f9924000.i2c: -->.000ms XFER_BEG msg_cnt:2
addr:0x20
<6>[   30.225990] i2c-msm-v2 f9924000.i2c:   0.000ms XFER_BUF msg[0] pos:0
adr:0x20 len:1 is_rx:0x0 last:0x0
<6>[   30.235277] i2c-msm-v2 f9924000.i2c:   0.001ms XFER_BUF msg[1] pos:0
adr:0x20 len:80 is_rx:0x1 last:0x1

//Received completion interrupt from controller
<6>[   30.314963] i2c-msm-v2 f9924000.i2c: 272.782ms  DONE_OK timeout-
used:560msec time_left:560msec
<6>[   30.323557] i2c-msm-v2 f9924000.i2c: 290.956ms ACTV_END ret:0
jiffies_left:10/100 read_cnt:0

//Transfer complete. Total time taken=290.958msms
```

<6>[30.331978] i2c-msm-v2 f9924000.i2c: 290.958ms XFER_END ret:2 err:[NONE] msgs_sent:2 BC:95 B/sec:326 i2c-stts:OK

5 SPI

This chapter describes the Serial Peripheral Interface (SPI) and explains how to configure it in the kernel.

5.1 Hardware overview

For a BLSP overview, see section *UART→Hardware overview*; for a QUP overview, see section *I2C→Hardware overview*.

5.1.1 SPI core

The SPI allows full-/half-duplex, synchronous, serial communication between a master and slave. There is no explicit communication framing, error checking, or defined data word length. Hence, the communication is strictly at the raw bit level.

Key features:

- Supports up to 50 MHz
- Supports 4 to 32 bits per word of transfer
- Supports a maximum of four Chip Selects (CS) per bus
- Supports BAM

5.1.2 QUP SPI parameters

To match the labeling in the software interface manual, each QUP is identified by a BLSP core (1 or 2) and QUP core (0 to 5). In hardware design documents, BLSPs are identified as BLSP[1:12].

MSM8916, MSM8936, MSM8909, and MDM9x35 chipsets contain a single BLSP core, so BLSP2_QUP0:5 is applicable only for the MSM8992 and MSM8994 chipsets.

Table 5-1 QUP physical address, IRQ numbers, Kernel SPI clock name, consumer, producer pipes, BLSP_BAM physical address, BAM IRQ number for MSM8994 and MSM8992

BLSP hardware ID	QUP core	Physical address (QUP_BASE_ADDRESS), size	IRQ #	Bus master ID	Kernel SPI clock name	Consumer, producer pipes	BLSP_BAM physical address, size, IRQ number
BLSP 1	BLSP1 QUP0	0xF9923000,0x500	95	86	clk_gcc_blsp1_qup1_spi_apps_clk	12, 13	0xF9904000, 0x19000, 238
BLSP 2	BLSP1 QUP1	0xF9924000,0x500	96	86	clk_gcc_blsp1_qup2_spi_apps_clk	14, 15	0xF9904000, 0x19000, 238
BLSP 3	BLSP1 QUP2	0xF9925000,0x500	97	86	clk_gcc_blsp1_qup3_spi_apps_clk	16, 17	0xF9904000, 0x19000, 238
BLSP 4	BLSP1 QUP3	0xF9926000,0x500	98	86	clk_gcc_blsp1_qup4_spi_apps_clk	18, 19	0xF9904000, 0x19000, 238
BLSP 5	BLSP1 QUP4	0xF9927000,0x500	99	86	clk_gcc_blsp1_qup5_spi_apps_clk	20, 21	0xF9904000, 0x19000, 238
BLSP 6	BLSP1 QUP5	0xF9928000,0x500	100	86	clk_gcc_blsp1_qup6_spi_apps_clk	22, 23	0xF9944000, 0x19000, 239
BLSP 7	BLSP2 QUP0	0xF9963000,0x500	101	84	clk_gcc_blsp2_qup1_spi_apps_clk	12, 13	0xF9944000, 0x19000, 239
BLSP 8	BLSP2 QUP1	0xF9964000,0x500	102	84	clk_gcc_blsp2_qup2_spi_apps_clk	14, 15	0xF9944000, 0x19000, 239
BLSP 9	BLSP2 QUP2	0xF9965000,0x500	103	84	clk_gcc_blsp2_qup3_spi_apps_clk	16, 17	0xF9944000, 0x19000, 239
BLSP10	BLSP2 QUP3	0xF9966000,0x500	104	84	clk_gcc_blsp2_qup4_spi_apps_clk	18, 19	0xF9944000, 0x19000, 239
BLSP11	BLSP2 QUP4	0xF9967000,0x500	105	84	clk_gcc_blsp2_qup5_spi_apps_clk	20, 21	0xF9944000, 0x19000, 239
BLSP12	BLSP2 QUP5	0xF9968000,0x500	106	84	clk_gcc_blsp2_qup6_spi_apps_clk	22, 23	0xF9944000, 0x19000, 239

Table 5-2 QUP physical address, IRQ numbers, Kernel SPI clock name, Consumer, producer pipes, BLSP_BAM physical address, BAM IRQ number for MSM8916, MSM8936, MSM8939, and MSM8909

BLSP hardware ID	QUP core	Physical address (QUP_BASE_ADDRESS)	IRQ #	Bus master ID	Kernel SPI clock name	Consumer, producer pipes	BLSP_BAM physical address, size, IRQ number
BLSP 1	BLSP1 QUP0	0x78B5000,0x600	95	86	clk_gcc_blsp1_qup1_spi_apps_clk	4, 5	0x7884000, 0x23000, 238
BLSP 2	BLSP1 QUP1	0x78B6000,0x600	96	86	clk_gcc_blsp1_qup2_spi_apps_clk	6, 7	0x7884000, 0x23000, 238
BLSP 3	BLSP1 QUP2	0x78B7000,0x600	97	86	clk_gcc_blsp1_qup3_spi_apps_clk	8, 9	0x7884000, 0x23000, 238
BLSP 4	BLSP1 QUP3	0x78B8000,0x600	98	86	clk_gcc_blsp1_qup4_spi_apps_clk	10, 11	0x7884000, 0x23000, 238
BLSP 5	BLSP1 QUP4	0x78B9000,0x600	99	86	clk_gcc_blsp1_qup5_spi_apps_clk	12, 13	0x7884000, 0x23000, 238
BLSP 6	BLSP1 QUP5	0x78BA000,0x600	100	86	clk_gcc_blsp1_qup6_spi_apps_clk	14, 15	0x7884000, 0x23000, 238

Table 5-3 QUP physical address, IRQ numbers, Kernel SPI clock name, consumer, producer pipes, BLSP_BAM physical address, BAM IRQ number for MSM8996

BLSP hardware ID	QUP core	Physical address (QUP_BASE_ADDRESS), size	IRQ #	Bus master ID	Kernel SPI clock name	Consumer, producer pipes	BLSP_BAM physical address, size, IRQ number
BLSP 1	BLSP1 QUP0	0x7575000,0x600	95	86	clk_gcc_blsp1_qup1_spi_apps_clk	12, 13	0x7544000, 0x2B000,238
BLSP 2	BLSP1 QUP1	0x7576000,0x600	96	86	clk_gcc_blsp1_qup2_spi_apps_clk	14, 15	0x7544000, 0x2B000,238
BLSP 3	BLSP1 QUP2	0x7577000,0x600	97	86	clk_gcc_blsp1_qup3_spi_apps_clk	16, 17	0x7544000, 0x2B000,238
BLSP 4	BLSP1 QUP3	0x7578000,0x600	98	86	clk_gcc_blsp1_qup4_spi_apps_clk	18, 19	0x7544000, 0x2B000,238
BLSP 5	BLSP1 QUP4	0x7579000,0x600	99	86	clk_gcc_blsp1_qup5_spi_apps_clk	20, 21	0x7544000, 0x2B000,238
BLSP 6	BLSP1 QUP5	0x757A000,0x600	100	86	clk_gcc_blsp1_qup6_spi_apps_clk	22, 23	0x7544000, 0x2B000,238
BLSP 7	BLSP2 QUP0	0x75B5000,0x600	101	84	clk_gcc_blsp2_qup1_spi_apps_clk	12, 13	0x7584000, 0x2B000,239
BLSP 8	BLSP2 QUP1	0x75B6000,0x600	102	84	clk_gcc_blsp2_qup2_spi_apps_clk	14, 15	0x7584000, 0x2B000,239
BLSP 9	BLSP2 QUP2	0x75B7000,0x600	103	84	clk_gcc_blsp2_qup3_spi_apps_clk	16, 17	0x7584000, 0x2B000,239
BLSP10	BLSP2 QUP3	0x75B8000,0x600	104	84	clk_gcc_blsp2_qup4_spi_apps_clk	18, 19	0x7584000, 0x2B000,239
BLSP11	BLSP2 QUP4	0x75B9000,0x600	105	84	clk_gcc_blsp2_qup5_spi_apps_clk	20, 21	0x7584000, 0x2B000,239
BLSP12	BLSP2 QUP5	0x75BA000,0x600	106	84	clk_gcc_blsp2_qup6_spi_apps_clk	22, 23	0x7584000, 0x2B000,239

Table 5-4 QUP physical address, IRQ numbers, Kernel SPI clock name, consumer, producer pipes, BLSP_BAM physical address, BAM IRQ number for MSM8952, MSM8953, MSM8956/76

NOTE: The following table has been updated.

BLSP hardware ID	QUP core	Physical address (QUP_BASE_ADDRESS), size	IRQ #	Bus master ID	Kernel SPI clock name	Consumer, producer pipes	BLSP_BAM physical address, size, IRQ number
BLSP 1	BLSP1 QUP0	0x78B5000,0x600	95	86	clk_gcc_blsp1_qup1_spi_apps_clk	4, 5	0x7884000, 0x1F000, 238
BLSP 2	BLSP1 QUP1	0x78B6000,0x600	96	86	clk_gcc_blsp1_qup2_spi_apps_clk	6, 7	0x7884000, 0x1F000, 238
BLSP 3	BLSP1 QUP2	0x78B7000,0x600	97	86	clk_gcc_blsp1_qup3_spi_apps_clk	8, 9	0x7884000, 0x1F000, 238
BLSP 4	BLSP1 QUP3	0x78B8000,0x600	98	86	clk_gcc_blsp1_qup4_spi_apps_clk	10, 11	0x7884000, 0x1F000, 238
BLSP 5	BLSP2 QUP0	0x7AF5000,0x600	299	84	clk_gcc_blsp2_qup1_spi_apps_clk	4, 5	0x7AC4000, 0x1F000, 239
BLSP 6	BLSP2 QUP1	0x7AF6000,0x600	300	84	clk_gcc_blsp2_qup2_spi_apps_clk	6, 7	0x7AC4000, 0x1F000, 239

BLSP 7	BLSP2 QUP2	0x7AF7000,0x600	301	84	clk_gcc_blsp2_qup3_spi_apps_clk	8, 9	0x7AC4000, 0x1F000, 239
BLSP 8	BLSP2 QUP3	0x7AF8000,0x600	302	84	clk_gcc_blsp2_qup4_spi_apps_clk	10, 11	0x7AC4000, 0x1F000, 239

Table 5-5 QUP physical address, IRQ numbers, Kernel SPI clock name, consumer, producer pipes, BLSP_BAM physical address, BAM IRQ number for MDM9x35

BLSP hardware ID	QUP core	Physical address (QUP_BASE_ADDRESS), size	IRQ #	Bus master ID	Kernel SPI clock name	Consumer, producer pipes	BLSP_BAM physical address, size, IRQ number
BLSP 1	BLSP1 QUP0	0xF9923000,0x500	95	86	clk_gcc_blsp1_qup1_spi_apps_clk	12, 13	0xF9904000, 0x19000, 238
BLSP 2	BLSP1 QUP1	0xF9924000,0x500	96	86	clk_gcc_blsp1_qup2_spi_apps_clk	14, 15	0xF9904000, 0x19000, 238
BLSP 3	BLSP1 QUP2	0xF9925000,0x500	97	86	clk_gcc_blsp1_qup3_spi_apps_clk	16, 17	0xF9904000, 0x19000, 238
BLSP 4	BLSP1 QUP3	0xF9926000,0x500	98	86	clk_gcc_blsp1_qup4_spi_apps_clk	18, 19	0xF9904000, 0x19000, 238
BLSP 5	BLSP1 QUP4	0xF9927000,0x500	99	86	clk_gcc_blsp1_qup5_spi_apps_clk	20, 21	0xF9904000, 0x19000, 238
BLSP 6	BLSP1 QUP5	0xF9928000,0x500	100	86	clk_gcc_blsp1_qup6_spi_apps_clk	22, 23	0xF9904000, 0x19000, 238

Table 5-6 QUP physical address, IRQ numbers, Kernel SPI clock name, consumer, producer pipes, BLSP_BAM physical address, BAM IRQ number for MDM9x40/9x45, MDM9x50/9x55

BLSP hardware ID	QUP core	Physical address (QUP_BASE_ADDRESS), size	IRQ #	Bus master ID	Kernel SPI clock name	Consumer, producer pipes	BLSP_BAM physical address, size, IRQ number
BLSP 1	BLSP1 QUP0	0x78B5000,0x600	95	86	clk_gcc_blsp1_qup1_spi_apps_clk	8, 9	0x7884000, 0x23000, 238
BLSP 2	BLSP1 QUP1	0x78B6000,0x600	96	86	clk_gcc_blsp1_qup2_spi_apps_clk	10, 11	0x7884000, 0x23000, 238
BLSP 3	BLSP1 QUP2	0x78B7000,0x600	97	86	clk_gcc_blsp1_qup3_spi_apps_clk	12, 13	0x7884000, 0x23000, 238
BLSP 4	BLSP1 QUP3	0x78B8000,0x600	98	86	clk_gcc_blsp1_qup4_spi_apps_clk	14, 15	0x7884000, 0x23000, 238

Table 5-7 QUP physical address, IRQ numbers, Kernel I2C clock name, consumer, producer pipes, BLSP_BAM physical address, BAM IRQ number for MDM9x07

NOTE: The following table was added to this document revision.

BLSP hardware ID	QUP core	Physical address (QUP_BASE_ADDRESS), size	IRQ #	Bus master ID	Kernel I2C clock name	Consumer, producer pipes	BLSP_BAM physical address, size, IRQ number
BLSP 1	BLSP1 QUP0	0x78B5000,0x1000	95	86	clk_gcc_blsp1_qup1_i2c_apps_clk	12, 13	0x7884000, 0x23000, 238
BLSP 2	BLSP1 QUP1	0x78B6000,0x1000	96	86	clk_gcc_blsp1_qup2_i2c_apps_clk	14, 15	0x7884000, 0x23000, 238
BLSP 3	BLSP1 QUP2	0x78B7000,0x1000	97	86	clk_gcc_blsp1_qup3_i2c_apps_clk	16, 17	0x7884000, 0x23000, 238
BLSP 4	BLSP1 QUP3	0x78B8000,0x1000	98	86	clk_gcc_blsp1_qup4_i2c_apps_clk	18, 19	0x7884000, 0x23000, 238
BLSP 5	BLSP1 QUP4	0x78B9000,0x1000	99	86	clk_gcc_blsp1_qup5_i2c_apps_clk	20, 21	0x7884000, 0x23000, 238
BLSP 6	BLSP1 QUP5	0x78BA000,0x1000	100	86	clk_gcc_blsp1_qup6_i2c_apps_clk	22, 23	0x7884000, 0x23000, 238

5.2 Configure kernel SPI

The SPI can operate in FIFO-based mode or Data Mover mode (BAM). If large amounts of data are to be transferred, enable BAM to offload the CPU. Additional fields are needed in the DTS node to enable SPI BAM mode. See section *SPI→SPI power management,* for detailed information.

5.2.1 Code changes

The following files are used to configure a QUP core as a SPI device in the kernel:

File type	Description
Device tree source	For MSM and APQ products: /kernel/arch/arm/boot/dts/qcom/<*chipset*>.dtsi Where <`chipset`> corresponds to the applicable chipset, for example: /kernel/arch/arm/boot/dts/qcom/msm8909.dtsi For MDM9x35: /kernel/arch/arm/boot/dts/msm9635.dtsi
Clock table	The clock nodes need to be added to the DTS file. /kernel/drivers/clk/qcom/clock-gcc-<chipset>.c
Pinctrl settings	The pin control table is located in the following file: /kernel/arch/arm/boot/dts/qcom/<*chipset*>-pinctrl.dtsi

This section describes the steps required to configure and use the BLSP1_QUP1 QUP core as an SPI master.

1. Create a device tree node for the SPI master.

   In the `/kernel/arch/arm/boot/dts/qcom/<chipset>.dtsi` file, add a new device tree node.

```
aliases{
   spi0 = &spi_0;
};

spi_0: spi@f9923000 { //QUP Base address for BLSP1_QUP1
  compatible = "qcom,spi-qup-v2"; //Manufacturer and Model
  cell-index = <0>; //BUS ID can be any #, use a large #
    interrupt-names = "spi_irq", "spi_bam_irq";

  /* Add BAM IRQ
        BLSP1: 238
        BLSP2: 239
      First Field: 0 SPI interrupt (Shared Peripheral Interrupt)
      Second Field: Interrupt #
      Third Field: Trigger type, keep 0  */
  interrupts = <0 95 0>, <0 238 0>;
  qcom,infinite-mode = <0>;
  qcom,use-bam; // Enable BAM mode
  /* Add BAM pipes, refer Table 5-1 to 5-x */
  qcom,bam-consumer-pipe-index = <12>;
  qcom,bam-producer-pipe-index = <13>;
  qcom,ver-reg-exists;
  spi-max-frequency = <5000000>; //Maximum supported frequency in HZ
  /* address,size for slave chips
     master-id = 86 for BLSP1
     84 for BLSP2
 Refer to BLSP Bus Master ID Table for more information.
 Master ID can be obtained from
 /kernel/arch/arm/boot/dts/msm8974-bus.dtsi */
  qcom,master-id = <86>;
  #address-cells = <1>;
  #size-cells = <0>;
      /* Assign AHB and Core clocks. clock regime considers qup[1:6]. So
      BLSP1 QUP0 maps to clk_gcc_blsp1_qup1_spi_apps_clk
      */
              clock-names = "iface_clk", "core_clk";
              clocks = <&clock_gcc clk_gcc_blsp1_ahb_clk>,
                    <&clock_gcc clk_gcc_blsp1_qup1_spi_apps_clk>;
```

Additional information	Location
Device tree	`/kernel/Documentation/devicetree/bindings/arm/gic.txt` `/kernel/Documentation/devicetree/bindings/spi/spi_qsd.txt`

2. Configure SPI master GPIOs by using pinctrl setting:
   a. Please refer to the chipset device specification to check BLSP gpio assignment, and refer to the chipset software interface to check BLSP gpio configuration.
   b. Open the `.dtsi` file located at:

      `kernel/arch/arm/boot/dts/qcom/<chipset>-pinctrl.dtsi`
   c. Modify the pin control settings as shown in the following example. For more information, refer to pin control documentation located at /kernel/Documentation/devicetree/bindings/pinctrl/msm-pinctrl.txt.

```
&soc {
  tlmm_pinmux: pinctrl@fd510000 {

//snip

spi0_active {
   qcom,pins = <&gp 0>, <&gp 1>, <&gp 2>, <&gp 3>;
   qcom,num-grp-pins = <4>;
   qcom,pin-func = <1>; // OEM must get the right gpio
func number from the software interface: TLMM chapter!
   label = "spi0_active";
   spi_0_active: default {
     drive-strength = <16>;
     bias-pull-up;
   };
 };

 spi0_sleep {
   qcom,pins = <&gp 0>, <&gp 1>, <&gp 2>, <&gp 3>;
   qcom,num-grp-pins = <4>;
   qcom,pin-func = <0>;
   label = "spi0_sleep";
   spi_0_sleep: sleep {
     drive-strength = <2>;
     bias-pull-up;
   };
 };
```

From kernel 3.14 onwards that MSM8996 is using, pinctrl framework changes a bit. Refer to the following demo to configure the SPI GPIOs:

```
&soc {
   tlmm: pinctrl@01010000 {
                compatible = "qcom,msm8996-pinctrl";

//snip
  spi_0 {
    spi_0_active: spi_0_active {
      spi_0 {
        pins = "gpio0", "gpio1", "gpio3";
        function = "blsp_spi1"; //For how to get the
function name, please check the pinctrl driver code at
kernel/drivers/pinctrl/qcom/pinctrl-msm8996.c

        drive-strength = <6>;
        bias-disable;
      };
    };

    spi_0_sleep: spi_0_sleep {
      spi_0 {
        pins = "gpio0", "gpio1", "gpio3";
        function = "blsp_spi1"; //For how to get the
function name, please check the pinctrl driver code at
kernel/drivers/pinctrl/qcom/pinctrl-msm8996.c

        drive-strength = <6>;
        bias-disable;
      };
    };
  };
```

3. Configure BAM pipes for the SPI master by modifying the TrustZone source code. See Step 3 in Section 3.4.1.

4. Verify configuration settings. If all the information was correctly entered, the SPI bus will be registered under `/sys/class/spi_master/spi#`, where cell-index matches the bus number.

```
adb shell --> Get adb shell
cd /sys/class/spi_master to list all the spi master
root@android:/sys/class/spi_master # ls
ls
spi0
spi6
spi7
```

5.2.2 SPI slave device sample code

When the SPI bus is registered, create a slave device driver and register it with the SPI master. For examples of SPI slave devices, refer to the following files:

```
/kernel/arch/arm/boot/dts/msm8994-cdp.dtsi
/kernel/Documentation/devicetree/bindings/spi/spi_qsd.txt
/kernel/Documentation/devicetree/bindings/spi/spi-bus.txt
```

The following procedure shows the minimum requirements for registering a slave device.

1. Create a device tree node for the SPI slave.
   a. Open the following file:

      ```
      /kernel/arch/arm/boot/dts/msm8994-cdp.dtsi
      ```

   b. Add the new device tree node:

```
spi@f9923000 {
  qcom-spi-test@0 { //Slave driver and CS ID
    compatible = "qcom,spi-test"; //Manufacture, and Model
    reg = <0>; //Same as CS ID
    spi-max-frequency = <50000000>;
    /*
     * Max Frequency for Device, it should not exceed the spi-max-
frequency defined in spi master driver's device tree.
     * Otherwise the spi-max-frequency defined in spi master will be
used.
    */

    /*
    * Following elements are optional.  For more detail respect to
    * SPI please see /kernel/Documentation/devicetree/bindings
      /spi/spi-bus.txt
    */
    spi-cpol; //CPOL bit set for SPI polarity
    spi-cpha; //CPHA bit set for SPI Phase
    spi-cs-high; //CS Active High
```

2. Create or modify the SPI slave device driver. The following provides an example of the slave driver.

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/delay.h>
#include <linux/spi/spi.h>
#include <linux/interrupt.h>
#include <linux/slab.h>
#include <linux/gpio.h>
#include <linux/debugfs.h>
#include <linux/seq_file.h>
#include <linux/regulator/consumer.h>
#include <linux/string.h>
#include <linux/of_gpio.h>

#ifdef CONFIG_OF //Open firmware must be defined for dts useage
static struct of_device_id qcom_spi_test_table[] = {
      { .compatible = "qcom,spi-test",}, //Compatible node must match
                                         //dts
      { },
};
#else
#define qcom_spi_test_table NULL
#endif


#define BUFFER_SIZE 4<<10
struct spi_message spi_msg;
struct spi_transfer spi_xfer;
u8 *tx_buf; //This needs to be DMA friendly buffer
static int spi_test_transfer(struct spi_device *spi)
{
      spi->mode |=SPI_LOOP;  //Enable Loopback mode
      spi_message_init(&spi_msg);

      spi_xfer.tx_buf = tx_buf;
      spi_xfer.len = BUFFER_SIZE;
      spi_xfer.bits_per_word = 8;

/*
 * Spi master driver will round the speed_hz to the lower nearest
frequency in the spi clock table ftbl_blsp*_qup*_spi_apps_clk_src[]
with kernel/drivers/clk/qcom/clock-gcc-(chipset).c
*/

      spi_xfer.speed_hz = spi->max_speed_hz;

      spi_message_add_tail(&spi_xfer, &spi_msg);

      return spi_sync(spi, &spi_msg);
}


static int spi_test_probe(struct spi_device *spi)
{
```

```
        //allocate memory for transfer
      tx_buf = kmalloc(BUFFER_SIZE, GFP_ATOMIC);
      if(tx_buf == NULL){
            dev_err(&spi->dev, "%s: mem alloc failed\n", __func__);
            return -ENOMEM;
      }
      //Parse data using dt.
      if(spi->dev.of_node){
        irq_gpio = of_get_named_gpio_flags(spi->dev.of_node,
"qcom_spi_test,irq-gpio", 0, NULL);
      }
      irq = spi->irq;
      cs = spi->chip_select;
      cpha = ( spi->mode & SPI_CPHA ) ? 1:0;
      cpol = ( spi->mode & SPI_CPOL ) ? 1:0;
      cs_high = ( spi->mode & SPI_CS_HIGH ) ? 1:0;
      max_speed = spi->max_speed_hz;
      dev_err(&spi->dev, "gpio [%d] irq [%d] gpio_irq [%d] cs [%x] CPHA
[%x] CPOL [%x] CS_HIGH [%x]\n",
            irq_gpio, irq, gpio_to_irq(irq_gpio), cs, cpha, cpol,
cs_high);

      dev_err(&spi->dev, "Max_speed [%d]\n", max_speed );

      //Transfer can be done after spi_device structure is created
      spi->bits_per_word = 8;
      dev_err(&spi->dev, "SPI sync returned [%d]\n",
spi_test_transfer(spi));
      return 0;
}
//SPI Driver Info
static struct spi_driver spi_test_driver = {
      .driver = {
            .name = "qcom_spi_test",
            .owner = THIS_MODULE,
            .of_match_table = qcom_spi_test_table,
      },
      .probe      = spi_test_probe,
};


static int __init spi_test_init(void)
{
      return spi_register_driver(&spi_test_driver);
}

static void __exit spi_test_exit(void)
{
      spi_unregister_driver(&spi_test_driver);
}


module_init(spi_test_init);
module_exit(spi_test_exit);
MODULE_DESCRIPTION("SPI TEST");
MODULE_LICENSE("GPL v2");
```

3. Verify the SPI slave device driver is configured and loaded successfully.

   In the kernel log, the following message indicates the device tree was successfully configured.

```
<3>[    2.503571] qcom_spi_test spi6.0: spi_test_probe
<3>[    2.507305] qcom_spi_test spi6.0: gpio [61] irq [306] gpio_irq
[306]
                       cs [0] CPHA [1] CPOL [1] CS_HIGH [1]
<3>[    2.516825] qcom_spi_test spi6.0: Max_speed [4800000]
<3>[    2.521932] qcom_spi_test spi6.0: SPI sync returned [0]

Because a SPI transfer has been triggered during the slave device driver
probe, we can use this slave device driver to verify the SPI master
transfer.
```

5.2.3 Configure and use multi-CS SPI

NOTE: This section was added to this document revision.

Only specified BLSP QUP cores of each platform supports multi-CS feature. Refer to device specification to figure out which QUP cores support this feature.

Consider MSM8994 platform as example, following pseudo code shows how to configure and use multi-CS feature:

1. From device specification, BLSP2(BLSP1_QUP1: 0xF9924000) supports multi-CS which has 4 CSs(CS0: GPIO43, CS1:GPIO_90, CS2: GPIO24, CS3:GPIO27)
2. Configure CS GPIO pins as SPI CS function in device tree and pinctrl:
   In kernel/arch/arm/boot/dts/qcom/msm8994-pinctrl.dtsi, define spi1_cs1_active/sleep, spi1_cs2_active/sleep, spi1_cs3_active/sleep refer to spi1_cs0
3. Include cs1, cs2 and cs3 pinctrl configuration in spi1's device tree:

   In kernel/arch/arm/boot/dts/qcom/msm8994.dtsi

   spi_1: spi@f9924000 { /* BLSP1 QUP1 */

   < snip >

   compatible = "qcom,spi-qup-v2";

   /* Add CS1, CS2, and CS3 pinctrl

   pinctrl-0 = <&spi1_default &spi1_cs0_active &spi1_cs1_active

   &spi1_cs2_active &spi1_cs3_active>;

   pinctrl-1 = <&spi1_sleep &spi1_cs0_sleep &spi1_cs1_sleep &spi1_cs1_sleep

   &spi1_cs2_sleep &spi1_cs3_sleep>;

   < snip >

   };

4. Configure and use multi-CS in spi slave device driver

```
spi@f9924000 {
spi_slave0@0 { //Slave driver and CS ID
compatible = "slave_0"; //Manufacture, and Model
        reg = <0>; //Same as CS ID, CS0
            .....
        };
spi_slave_1@1 {
        //Slave driver and CS ID
        compatible = "slave_1"; //Manufacture, and Model
reg = <1>; //Same as CS ID, CS1
            .....
        };
spi_slave_2@2 {
//Slave driver and CS ID
compatible = "slave_1"; //Manufacture, and Model
reg = <2>; //Same as CS ID, CS2
            .....
        };
spi_slave_3@3 {
//Slave driver and CS ID
compatible = "slave_1"; //Manufacture, and Model
reg = <3>; //Same as CS ID, CS3
            .....
};
};
```

5.2.4 Debug high-speed SPI

1. If FIFO mode SPI works but BAM mode SPI does not work, ensure that the application processor can access designated pipes.the
2. MDM9x35 only has a BLSP1 core. Only BLSP1_P addresses in Table 5-8 are applicable.

Table 5-8 BLSP QUP BAM pipe physical address for MSM8994 and MSM8992

BLSP pipe number	PIPE_TRUST_REG	PIPE_CTRL_REG
BLSP1_P 12	0xF9911030	0xF9911000
BLSP1_P 13	0xF9912030	0xF9912000
BLSP1_P 14	0xF9913030	0xF9913000
BLSP1_P 15	0xF9914030	0xF9914000
BLSP1_P 16	0xF9915030	0xF9915000
BLSP1_P 17	0xF9916030	0xF9916000
BLSP1_P 18	0xF9917030	0xF9917000
BLSP1_P 19	0xF9918030	0xF9918000
BLSP1_P 20	0xF9919030	0xF9919000
BLSP1_P 21	0xF991A030	0xF991A000
BLSP1_P 22	0xF991B030	0xF991B000
BLSP1_P 23	0xF991C030	0xF991C000
BLSP2_P 12	0xF9951030	0xF9951000
BLSP2_P 13	0xF9952030	0xF9952000
BLSP2_P 14	0xF9953030	0xF9953000
BLSP2_P 15	0xF9954030	0xF9954000
BLSP2_P 16	0xF9955030	0xF9955000
BLSP2_P 17	0xF9956030	0xF9956000
BLSP2_P 18	0xF9957030	0xF9957000
BLSP2_P 19	0xF9958030	0xF9958000
BLSP2_P 20	0xF9959030	0xF9959000
BLSP2_P 21	0xF995A030	0xF995A000
BLSP2_P 22	0xF995B030	0xF995B000
BLSP2_P 23	0xF995C030	0xF995C000

Table 5-9 BLSP QUP BAM pipe physical address for MSM8916, MSM8909, and MSM8936

BLSP pipe number	PIPE_TRUST_REG	PIPE_CTRL_REG
BLSP1_P 4	0x07886030	0x0789B000
BLSP1_P 5	0x07886034	0x0789C000
BLSP1_P 6	0x07886038	0x0789D000
BLSP1_P 7	0x0788603C	0x0789E000
BLSP1_P 8	0x07886040	0x0789F000
BLSP1_P 9	0x07886044	0x078A0000
BLSP1_P 10	0x07886048	0x078A1000
BLSP1_P 11	0x0788604C	0x078A2000
BLSP1_P 12	0x07886050	0x078A3000
BLSP1_P 13	0x07886054	0x078A4000
BLSP1_P 14	0x07886058	0x078A5000
BLSP1_P 15	0x0788605C	0x078A6000

3. After the system boots up, attach a JTAG to the RPM and Krait cores.

```
t32 cmd: sys.m.a
```

4. Break all the Krait cores.

```
t32 cmd: b
```

5. Turn on the BLSP_AHB_CLK by ensuring that bits 17 and 15 are set for the APCS_CLOCK_BRANCH_ENA_VOTE register.

```
APCS_CLOCK_BRANCH_ENA_VOTE = 0xFC401484
```

6. Ensure that BLSP_PIPE_TRUST_REG is set to zero for the apps processor. Read using Secure mode.

   For example, for MSM8994:

   - BLSP1 pipe 22:

     ```
     D AZ:0xF991B030
     ```

   - BLSP1 pipe 23:

     ```
     D AZ:0xF991C030
     ```

7. Ensure that you can read/write to PIPE_CTRL_REG in Nonsecure mode.

   For example, for MSM8994:

   - Writing pipe 22:

     ```
     D.S A:0x0:0xF991B000 %LE %LONG 0xABCD
     ```

   - Writing pipe 23:

     ```
     D.S A:0x0:0xF991C000 %LE %LONG 0xABCD
     ```

   - Reading pipe 22:

     ```
     data.in A:0xF991B000 /long
     ```

   - Reading pipe 23:

     ```
     data.in A:0xF991C000 /long
     ```

8. If the above test fails, check the TrustZone modifications.

5.3 SPI power management

SPI slave devices must register system suspend/resume (SYSTEM_PM_OPS) handlers with the power management framework to ensure that no SPI transactions are initiated after the SPI master is suspended. For examples, see section *I2C→I2C power management*.

5.4 Steps to measure SPI throughput

NOTE: This section was added to this document revision.

OEMs can use spidev driver, spitest and ftrace combined together to measure SPI throughput.

By default, the spidev driver is compiled as a module, named, and is located at:

system/lib/modules/spidev.ko

The spidevtest source code locates at: (android_root)/vendor/qcom/proprietary/kernel-tests/spi

Compiled binary locates at: /data/kernel-tests/spidevtest

Following the steps below to execute the SPI throughput test, based on MSM8996 platform, other platforms may have slight difference in the folder path:

1. Find how many spi masters are configured in your system with the following command:

```
adb root
adb wait-for-device
adb shell
# cd /sys/class/spi_master
#ls  → each spi master will be listed as a folder under sysfs, in my system only one spi master, which is spi0
root@msm8996:/sys/class/spi_master # ls
ls
spi0
```

2. Load the spidev driver module with following command:

```
In adb shell:
# cd /system/lib/modules
# insmod spidev.ko busnum=0 chipselect=1 maxspeed=50000000
busnum = 0 means spi0,  chipselect=1 means we use cs 1, maxspeed= 50M means max spi transfer speed is limited as 50M.
Attention please,  maxspeed must be less than the maxspeed defined in spi master’s device tree resource, or you will encounter following error:
---------------- kernel log --------
spi_qsd 7575000.spi: Invalid transfer: 50000000 Hz, 32 bpw
 tx=ffffffc042600000, rx=ffffffc03e280000
----------------------------------------
```

If following error shows up after the above insmod command, it means your system image does not come from the same build with the boot image. As new kernel has the security feature, it will verify the security related key when loaded a module.

Solution: Recompile the spidev.ko, using the same environment which you are using to compile your boot image, and push the new built spidev.ko to /system/lib/modules

3. Enable the necessary ftrace mask of SPI driver with following command:

```
echo "" > /d/tracing/set_event
echo "" > /d/tracing/trace

echo 1 > /d/tracing/events/spi/spi_message_submit/enable
echo 1 > /d/tracing/events/spi/spi_message_start/enable
echo 1 > /d/tracing/events/spi/spi_message_done/enable
echo 1 > /d/tracing/events/spi/spi_transfer_start/enable
echo 1 > /d/tracing/events/spi/spi_transfer_stop/enable

 delta (spi_message_start, spi_message_done) to get the xfer time.
```

4. Use spidevtest to execute SPI transfer with following command:

```
In adb shell：
     cd /data/kernel-tests/spi
chmod 777 spidevtest
# ./spidevtest spidev0.1 -l -t 4 -z 50000000
./spidevtest spidev0.1 -l -t 4 -z 50000000
Test passed
```

5. Collect the ftrace log to calculate the throughput:

```
cat /d/tracing/trace
---- Trace log ------
spidevtest-3902  [001] d..3   134.138392: spi_message_submit: spi0.1
ffffffc0b9d77da8
spi0-243   [001] ...1   134.138474: spi_message_start: spi0.1
ffffffc0b9d77da8
spi0-243   [001] ...1   134.138512: spi_transfer_start: spi0.1
ffffffc0b9d4a480 len=32808
spi0-243   [000] ...1   134.147282: spi_transfer_stop: spi0.1
ffffffc0b9d4a480 len=32808
spi0-243   [000] ...1   134.147357: spi_message_done: spi0.1
ffffffc0b9d77da8 len=32808/32808
spidevtest-3902  [001] d..3   134.147985: spi_message_submit: spi0.1
ffffffc0b9d77da8
 spi0-243   [000] ...1   134.148275: spi_message_start: spi0.1
ffffffc0b9d77da8
 spi0-243   [000] ...1   134.148326: spi_transfer_start: spi0.1
ffffffc0b9d4a480 len=32808
 spi0-243   [000] ...1   134.156654: spi_transfer_stop: spi0.1
ffffffc0b9d4a480 len=32808
 spi0-243   [000] ...1   134.156715: spi_message_done: spi0.1
ffffffc0b9d77da8 len=32808/32808
 spidevtest-3902  [000] d..3   134.157526: spi_message_submit: spi0.1
ffffffc0b9d77da8
```

```
  spi0-243   [000] ...1   134.157643: spi_message_start: spi0.1
ffffffc0b9d77da8
   spi0-243   [000] ...1   134.157685: spi_transfer_start: spi0.1
ffffffc049119580 len=32808
    spi0-243   [000] ...1   134.166033: spi_transfer_stop: spi0.1
ffffffc049119580 len=32808
    spi0-243   [000] ...1   134.166095: spi_message_done: spi0.1
ffffffc0b9d77da8 len=32808/32808
```

6. Use the following formula to calculate the throughput:

Package_size * 8 / (1000*Delta_time) Unit: Mbps.
Detlta time (Unit : ms) : time delta between spi_message_start and spi_message_done

Example: based on the log we collected in step 5.

Delta_time_1: 134.147357 - 134.138474 = 8.883 ms

Delta_time_2: 134.156715 - 134.148275 = 8.44 ms

Delta_time_3: 134.166095 - 134.157643 = 8.452ms

Average_Delta = (8.883 + 8.44 + 8.452)/3 = 8.5916ms

Throughput is: 32808*8 / (8.5916*1000) = 30.55 Mbps.

To achieve best SPI throughput, following tips are suggested:

a. SPI transfer parameter setting: N=32bits per word, transfer speed: 50 MHz, large payload size (32K payload we are using during test).

b. Set Linux OS governor to performance mode, and better disable idle power collapse.

5.5 Code walkthrough

5.5.1 Probing

5.5.1.1 Call the SPI master probe

Similar to the UART probe, the SPI master probe is called with the following call stack. For details, see *UART→Code walkthrough-High-speed UART driver→Probing*.

```
-000|msm_spi_probe()
-001|platform_drv_probe()
-002|driver_probe_device()
-003|__driver_attach()
-004|bus_for_each_dev()
-005|bus_add_driver()
-006|driver_register()
-007|platform_driver_probe()
-008|do_one_initcall()
```

The following resources must be defined for successful SPI master registration.

Table 5-10 SPI master registration resources required for BAM

Resource	Description
msm_spi_dt_to_pdata--> msm_spi_dt_to_pdata_populate()	Parses the device tree
msm_spi_bam_get_resources	Gets BAM informations
msm_spi_request_irq	Gets IRQ information

Table 5-11 Device tree and clock resources required for SPI BAM

Resource	Description
Device tree	
spi-max-frequency	Maximum BUS frequency
qcom,master-id	BUS Scale ID
spi_physical	BLSP QUP base
spi_irq	QUP IRQ
If BAM is required	
qcom,use-bam	Enable BAM mode
qcom,bam-consumer-pipe-index	Consumer pipe index
qcom,bam-producer-pipe-index	Producer pipe index
spi_bam_physical	BLSP_BAM_BASE
spi_bam_irq	BLSP_BAM IRQ
Clock table	
core_clk	QUP core clock
baseaddress.spi	QUP core clock
iface_clk	AHB clock
baseaddress.spi	AHB clock

GPIOs must be properly defined in `board-<chipset>-gpiomux.c`.

5.5.1.2 Register the SPI master

Calling the `spi_register_master()` function from the probe registers the current master controller with the Linux SPI framework.

```
int spi_register_master(struct spi_master *master)
{
      static atomic_t          dyn_bus_id = ATOMIC_INIT((1<<15) - 1);
      struct device            *dev = master->dev.parent;
      struct boardinfo   *bi;
      int                status = -ENODEV;
      int                dynamic = 0;

        /* Each bus will be labeled as spi#*/
      dev_set_name(&master->dev, "spi%u", master->bus_num);
      status = device_add(&master->dev);

        ...
      /* If we're using a queued driver, start the queue */
      if (master->transfer)
            dev_info(dev, "master is unqueued, this is deprecated\n");
      else {
            status = spi_master_initialize_queue(master);
            if (status) {
                  device_unregister(&master->dev);
                  goto done;
            }
      }

     /* spi_master_list contain list of SPI masters that are registered */
      list_add_tail(&master->list, &spi_master_list);

      /* Register SPI devices from the device tree */
      of_register_spi_devices(master);
}
```

5.5.1.3 Register SPI slave

After the SPI master is registered by `spi_register_master()`, the slave probe is called.

```
-000|spi_test_probe() //SPI Slave Probe function
-001|spi_drv_probe()
-002|driver_probe_device()
-003|bus_for_each_drv()
-004|device_attach()
-005|bus_probe_device()
-006|device_add()
-007|spi_add_device()
-008|of_register_spi_devices()
-009|spi_register_master()
-010|msm_spi_probe() //SPI Master Probe
-011|platform_drv_probe()
-012|driver_probe_device()
```

```
-013|__driver_attach()
-014|bus_for_each_dev()
-015|bus_add_driver()
-016|driver_register()
-017|platform_driver_probe()
```

The slave probe has following prototype:

```
int(*probe)(struct spi device *spi)
```

When the slave device driver has an spi_device pointer, the slave device is free to initiate an SPI transfer as long as the SPI master is not in a Suspended state.

5.5.2 SPI transfer

5.5.2.1 Message structure

Figure 5-1 shows how SPI transactions are queued.

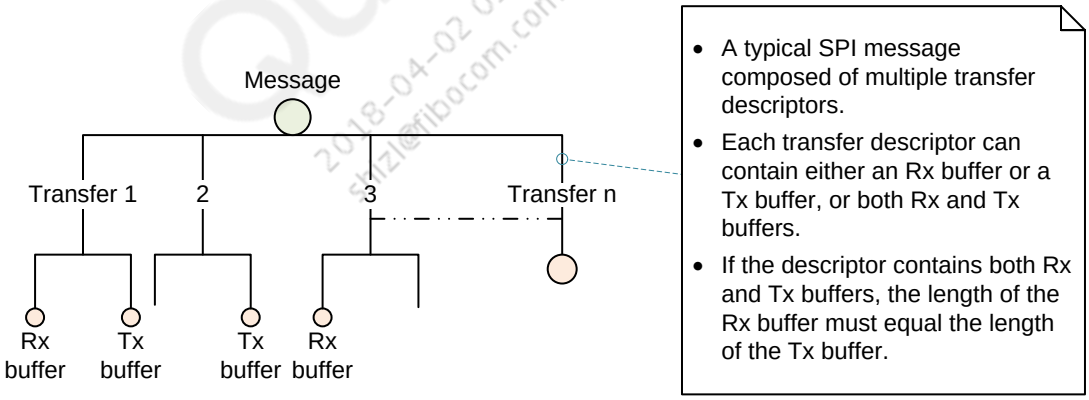


Figure 5-1 SPI message queue

For each `spi_sync()` or `spi_async()` function, a single message is processed.

5.5.2.2 spi_sync()

The `spi_sync()` function is a blocking call that waits until an entire message is transferred before returning to the caller.

```
int spi_sync(struct spi_device *spi, struct spi_message *message,
)
{
  DECLARE_COMPLETION_ONSTACK(done);
  int status;
  struct spi_master *master = spi->master;
  /* Initialize the completion call back */
  message->complete = spi_complete;
  message->context = &done;

  /* Queue the message */
  status = spi_async_locked(spi, message);

  /* Wait for completion signal from master */
  if (status == 0) {
    wait_for_completion(&done);
    status = message->status;
  }
  return status;
}
```

5.5.2.3 spi_async()

The `spi_async()` function is a nonblocking call that can also be called from an atomic context. With this function, a slave device can queue multiple messages and wait for the master to call back. For each message that is complete, the master calls the callback.

```
static int spi_async(struct spi_device *spi, struct spi_message *message)
{
  struct spi_master *master = spi->master;

  message->spi = spi;
  message->status = -EINPROGRESS;
  /* Queue the Transfer with the SPI Master */
  return master->transfer(spi, message);}
```

6 BLSP BAM

This chapter describes the BAM software architecture that is relevant to the BLSP. To understand the BLSP BAM, see the *Low-Speed Peripherals Overview* (80-NA157-24).

6.1 Source code

The `/kernel/arch/arm/mach-msm/include/mach/sps.h}` header file contains all of the functions, flags, and data structures that are exposed to client drivers.

The source directory is `/kernel/drivers/platform/msm/sps/}`.

6.2 Key functions

6.2.1 sps_phy2h()

This function checks the registered BAM device list, sps->bam_q, to see if a physical address of the BAM is already registered. If a BAM address is registered, this function returns the BAM handler, struct sps_bam.

6.2.2 sps_register_bam_device()

If the BAM device is not already registered, this function registers it with the BAM driver.

- Initializes the sps_bam structure by calling `sps_bam_device_init()`.
- Adds the sps_bam structure to the sps->bam_q list.
- Returns the handler for the sps_bam structure.

6.2.3 sps_alloc_endpoint()

This function allocates the sps_pipe structure and returns the handler after initializing it by calling `sps_client_init()`.

- Sets sps_pipe.client_state to SPS_STATE_DISCONNECT
- Sets sps_pipe.connect to SPSRM_CLEAR

6.2.4 sps_connect()

This function initializes the BAM hardware and estabishes communication between the BAM and processor.

- Copies the sps_connect structure to sps_pipe.connect
- Allocates the sps_connection structure and maps it to sps_pipe

- Configures and enables the BAM pipe
- Sets a connection from sps_pipe.client_state to SPS_STATE_CONNECT

6.2.5 sps_register_event()

This function registers an event handler for the sps_event by updating sps_pipe.event_regs.

6.2.6 sps_transfer_one()

This function queues a single descriptor into the BAM pipe by calling sps_bam_pipe_transfer_one.

- Updates sps_pipe.sys.desc_offset to the next location
- PIPE_EVENT_REG = "next_write"}

6.2.7 bam_isr()

This function is the ISR handler for the BLSP BAM.

- Determines which pipe caused an interrupt by reading the BAM_IRQ_SRCS register
- Calls pipe_handler-->pipe_handler_eot to process the interrupt
- Updates sps_pipe.sys.acked_offset with SW_DESC_OFST

Call stack:

```
-000|client_callback()
-001|trigger_event.isra.1()
-002|pipe_handler_eot()
-003|pipe_handler()
-004|bam_isr()
-005|handle_irq_event_percpu()
-006|handle_irq_event()
-007|handle_fasteoi_irq()
-008|generic_handle_irq()
-009|handle_IRQ()
-010|gic_handle_irq()
```

6.2.8 sps_disconnect()

This function disables the BAM hardware connection and deallocates any resources allocated by the SPS driver.

6.3 Key data structures

6.3.1 sps_drv * sps

This data structure is the global data structure.

```
struct sps_drv {
  /* Driver is ready */
  int is_ready;

  /* BAM devices */
  struct list_head bams_q;
};
```

6.3.2 sps_bam

This data structure stores BAM peripheral information.

```
struct sps_bam {

  /* BAM device properties, including connection defaults */
  struct sps_bam_props props;

  /* BAM device state */
  u32 state;

  /* Pipe state */
  u32 pipe_active_mask;
  u32 pipe_remote_mask;
  struct sps_pipe *pipes[BAM_MAX_PIPES];
  struct list_head pipes_q;

};
```

6.3.3 sps_pipe

This data structure stores BAM pipe information.

```
struct sps_pipe {
  /* Client state */
  u32 client_state;

  /* Connection states*/
  struct sps_connect connect;
  const struct sps_connection *map;

  /* Pipe parameters */
  u32 state;
  u32 pipe_index;
  u32 pipe_index_mask;
  u32 irq_mask;

  u32 num_descs; /* Size (number of elements) of descriptor FIFO */
  u32 desc_size; /* Size (bytes) of descriptor FIFO */

  /* System mode control */
  struct sps_bam_sys_mode sys;
};
```

6.3.4 struct sps_connect

This data structure stores pipe configuration data from the client.

```
struct sps_connect {
  /* Pipe configuration info */
  u32 source;
  u32 src_pipe_index;
  u32 destination;
  u32 dest_pipe_index;
  enum sps_mode mode;

  /* Connection Options*/
  enum sps_option options;

  /* Descriptor memory */
  struct sps_mem_buffer desc;
};
```

6.3.5 sps_register_event

This data structure stores information with respect to the event handler.

```
struct sps_register_event {
  /* Options that will trigger */
  enum sps_option options;
  enum sps_trigger mode;
  /* Handler or completion signal */
  struct completion *xfer_done;
  void (*callback)(struct sps_event_notify *notify);
  void *user;
};
```

6.3.6 sps_bam_sys_mode

This data structure stores descriptor buffer information and event offsets.

```
struct sps_bam_sys_mode {
  /* Descriptor FIFO control */
  u8 *desc_buf; /* Descriptor FIFO for BAM pipe */
  u32 desc_offset; /* Next new descriptor to be written to hardware */
  u32 acked_offset; /* Next descriptor to be retired by software */

  /* Descriptor cache control (!no_queue only) */
  u8 *desc_cache; /* Software cache of descriptor FIFO contents */
  u32 cache_offset; /* Next descriptor to be cached (ack_xfers only) */

};
```

7 GPIO

Each MSM/MDM/APQ chipset has a dedicated number of GPIOs that can be configured for multiple functions. For example, if you check the GPIO mapping for MSM8994 GPIO 0, you will see that the GPIO can be configured as one of the following functions at any time:

- Function 0 – GPIO
- Function 1 – BLSP1 SPI MOSI
- Function 2 – BLSP1 UART TX
- Function 3 – BLSP1 UIM data
- Function 4 – HDMI_RCV_DET

7.1 Critical registers

This section describes critical hardware registers that are important for debugging.

Refer to the software interface for detailed information on TLMM registers.

7.1.1 TLMM_GPIO_CFGn

TLMM_GPIO_CFGn controls the GPIO properties such as Output Enable, Drive Strength, Pull, and GPIO Function Select.

For example, for MSM8994:

```
Physical Address of TLMM_GPIO_CFGn = 0xFD511000 + (0x10 * n)
  n = GPIO #
  Example Address:
    0xFD511000 = GPIO_CFG0,  It's the GPIO0's configuration register
    0xFD511010 = GPIO_CFG1,  It's the GPIO1's configuration register
Bit definition for GPIO_CFGn
  Bits 31:11  Reserved
  Bit  10     GPIO_HIHYS_EN    Control the hihys_EN for GPIO
  Bit  9      GPIO_OE          Controls the Output Enable for GPIO
                               when in GPIO mode.
  Bits 8:6    DRV_STRENGTH     Control Drive Strength
                               000:2mA  001:4mA  010:6mA  011:8mA
                               100:10mA 101:12mA 110:14mA 111:16mA
  Bits 5:2    FUNC_SEL         Make sure Function is BLSP
                               Check Device Pinout for Correct Function
  Bits 1:0    GPIO_PULL        Internal Pull Configuration
                               00:No Pull   01: Pull Down
                               10:Keeper    11: Pull Up
```

7.1.2 TLMM_GPIO_IN_OUTn

TLMM_GPIO_IN_OUTn controls the output value or reads the current GPIO value.

```
Physical Address of TLMM_GPIO_IN_OUTn = 0xFD511004 + (0x10 * n)
  n = GPIO #
  Example Address:
    0xFD511004 = GPIO_IN_OUT0
    0xFD511014 = GPIO_IN_OUT1

Bit definition for GPIO_CFGn
  Bits 31:2  Reserved
  Bit  1     GPIO_OUT       Control value of the GPIO Output
  Bit  0     GPIO_IN        Allow you to read the Input value of the
                            GPIO
```

7.1.3 TLMM_GPIO_INTR_CFGn

TLMM_GPIO_INTR_CFGn controls the GPIO interrupt configuration settings.

```
Physical Address of TLMM_GPIO_INTR_CFGn = 0xFD511008 + (0x10 * n)
  n = GPIO #
  Example Address:
    0xFD511008 = GPIO_INTR_CFG0
    0xFD511018 = GPIO_INTR_CFG1

Bit definition for GPIO_CFGn
  Bits 31:9   Reserved
  Bit  8      DIR_CONN_IN          Being used as Direct Connect Interrupt.
                                    0: Default direct connect
                                    1: Enable Direct connect
  Bits 7:5    TARGET_PROC          Determine which processor a summary
                                   interrupt should get routed to.
                                    0x4: Apps Summary Interrupt
  Bit  4      INTR_RAW_STATUS_EN   Enable the RAW status for summary
interrupt.
                                    0: Disable
                                    1: Enable
  Bits 3:2    INTR_DECT_CTL        Control the Edge or Level Detection
                                    0x0: LEVEL sensitive
                                    0x1: Positive Edge
                                    0x2: Negative Edge
                                    0x3: Dual Edge
  Bit  1      INTR_POL_CTL         Control the Polarity Detection
                                    0x0: Active Low
                                    0x1: Active High
```

```
  Bits 0      INTR_ENABLE          Control if this GPIO generate summary
interrupt.
                                    0: Disable
                                    1: Enable
```

7.1.4 TLMM_GPIO_INTR_STATUSn

TLMM_GPIO_INTR_STATUSn indicates the status of the summary interrupt.

```
Physical Address of TLMM_GPIO_INTR_STATUSn = 0xFD51100C + (0x10 * n)
  n = GPIO #
  Example Address:
    0xFD51100C = GPIO_INTR_STATUS0
    0xFD51101C = GPIO_INTR_STATUS1

Bit definition for GPIO_CFGn
  Bits 31:1   Reserved
  Bit  0      INTR_STATUS          When read it return status of interrupt.
                                    0: No interrupt
                                    1: Pending Interrupt
```

7.1.5 Configure GPIOs in Linux kernel

This section describes the steps required to configure MSM8994 GPIOs in the Linux kernel. See Documentation/devicetree/bindings/pinctrl/msm-pinctrl.txt for more detailed instructions.

As an example, consider the Synaptics Touchscreen driver, which uses one I2C and two software-controlled MSM GPIOs, as indicated in Table 7-1.

Table 7-1 TS driver GPIOs in MSM8994

GPIO	Function	Pull settings		Drive strength/Vin	
		Active	Sleep	Active	Sleep
MSM_GPIO_61	Interrupt input	Pull-up	Pull-none	16 mA	16 mA
MSM_GPIO_60	Digital output	Pull-up	Pull-none	16 mA	16 mA

For MSM GPIO settings, see TLMM_GPIO_CFGn.

The following steps explain how to configure them.

7.1.6 Define pin controller node in DTS

For example, for MSM8994, add the pin controller nodes in `msm8994-pinctrl.dtsi`.

```
&soc {
        tlmm_pinmux: pinctrl@fd510000 {

                compatible = "qcom,msm-tlmm-8994";

                /* Base address and size of TLMM CSR registers */
                reg = <0xfd510000 0x4000>;

                /* First Field: 0 SPI interrupt (Shared Peripheral
Interrupt)
              Second Field: Interrupt #
              Third field: Trigger type, keep 0  */
                interrupts = <0 208 0>;
                <SNIP>
                        pmx_ts {
                        qcom,pins = <&gp 60>, <&gp 61>;
                        qcom,pin-func = <0>;
                        qcom,num-grp-pins = <2>;
                        label = "pmx_ts";

                        ts_active: ts_active {
                                drive-strength = <16>;
                                bias-pull-up;
                        };

                        ts_suspend: ts_suspend {
                                drive-strength = <16>;
                                bias-disable;
                        };
                        };
                <SNIP>
                };
        };
```

Add the above defined nodes to client node (synaptics_dsx_i2c) in `msm8994-cdp.dtsi`.

```
&soc {
<SNIP>
                i2c@f9924000 {
                synaptics@20 {
                        compatible = "synaptics,dsx"; //Manufacturer
and Model

                        reg = <0x20>; //Slave address
                        interrupt-parent = <&msm_gpio>;
        /* First Field: Interrupt #
        Second field: Trigger type
        bits[3:0] trigger type and level flags.
                1 = low-to-high edge triggered
                2 = high-to-low edge triggered
                4 = active high level-sensitive
                8 = active low level-sensitive
        bits[15:8] PPI interrupt cpu mask.  Each bit corresponds to
each of the 8 possible cpus attached to the GIC.  A bit set to '1'
indicated the interrupt is wired to that CPU.  Only valid for PPI
interrupts.  */                         interrupts = <61 0x2008>;
                        vdd-supply = <&pm8994_l14>;
                        avdd-supply = <&pm8994_l22>;
                        pinctrl-names = "pmx_ts_active",
"pmx_ts_suspend";
                        pinctrl-0 = <&ts_active>;
                        pinctrl-1 = <&ts_suspend>;
/* The following fields are defined in Synaptics TS driver.
Likewise, other I2C Slave drivers can define their own DTS fields.*/
                        synaptics,display-coords = <0 0 1599 2559>;
                        synaptics,panel-coords = <0 0 1599 2703>;
                        synaptics,reset-gpio = <&msm_gpio 60 0x00>;
                        synaptics,irq-gpio = <&msm_gpio 61 0x2008>;
                        synaptics,disable-gpios;
                };
            };
        <SNIP>
        };
```

7.1.7 Accessing GPIOs in driver

Using pinctrl information in the kernel driver (see synaptics_dsx_i2c.c), do the following:

1. In probe function get pinctrl from pinctrl.dtsi

```
ts_pinctrl = devm_pinctrl_get((platform_device->dev.parent));
```

2. In probe function get GPIO's different state settings.

```
pinctrl_state_active = pinctrl_lookup_state(ts_pinctrl,
"pmx_ts_active");


pinctrl_state_suspend = pinctrl_lookup_state(ts_pinctrl,
"pmx_ts_suspend");
```

3. Request the GPIO.

```
gpio_request(platform_data->irq_gpio, "rmi4_irq_gpio");
```

4. Set the GPIO direction.

```
gpio_direction_output(platform_data->reset_gpio, 1);
gpio_direction_input(platform_data->irq_gpio);
```

5. If it is an interrupt pin, request the IRQ.

```
int irqn = gpio_to_irq(platform_data->irq_gpio);
```

6. If it is a wakable interrupt then configure as such

```
enable_irq_wake(irqn);
```

7. Set different GPIO states when needed.

```
pinctrl_select_state(ts_pinctrl, pinctrl_state_active);
pinctrl_select_state(ts_pinctrl, pinctrl_state_suspend);
```

8. Write a value (HIGH/LOW) to output the GPIO.

```
gpio_set_value(platform_data->reset_gpio, 1);
gpio_set_value(platform_data->reset_gpio, 0);
```

9. Read the GPIO status.

```
int value = gpio_get_value(platform_data->irq_gpio);
```

7.2 Callflow for GPIO interrupt

Figure 7-1 through Figure 7-3 show the call flow for registering and firing a GPIO interrupt.

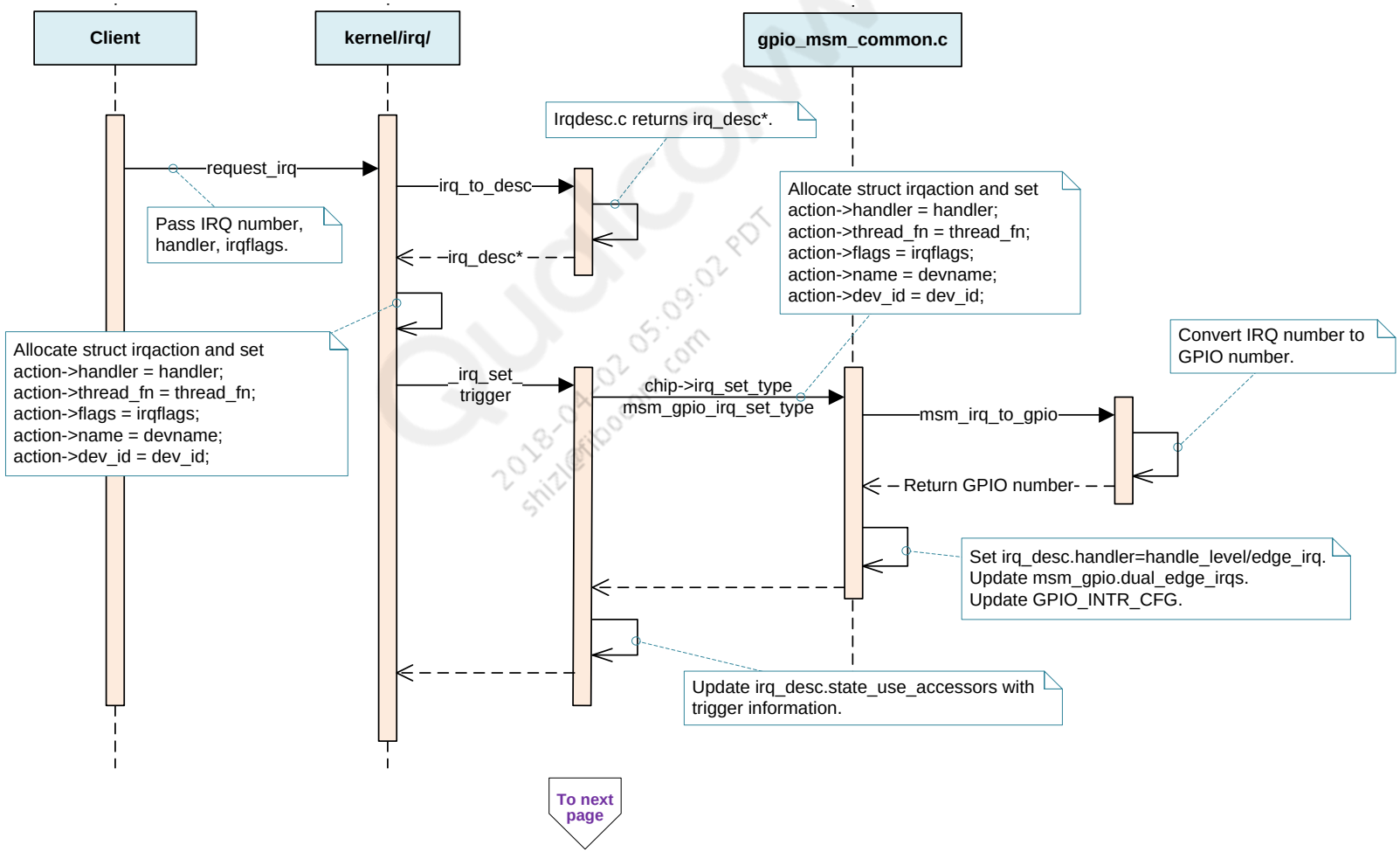


Figure 7-1 Register a GPIO IRQ (1 of 2)

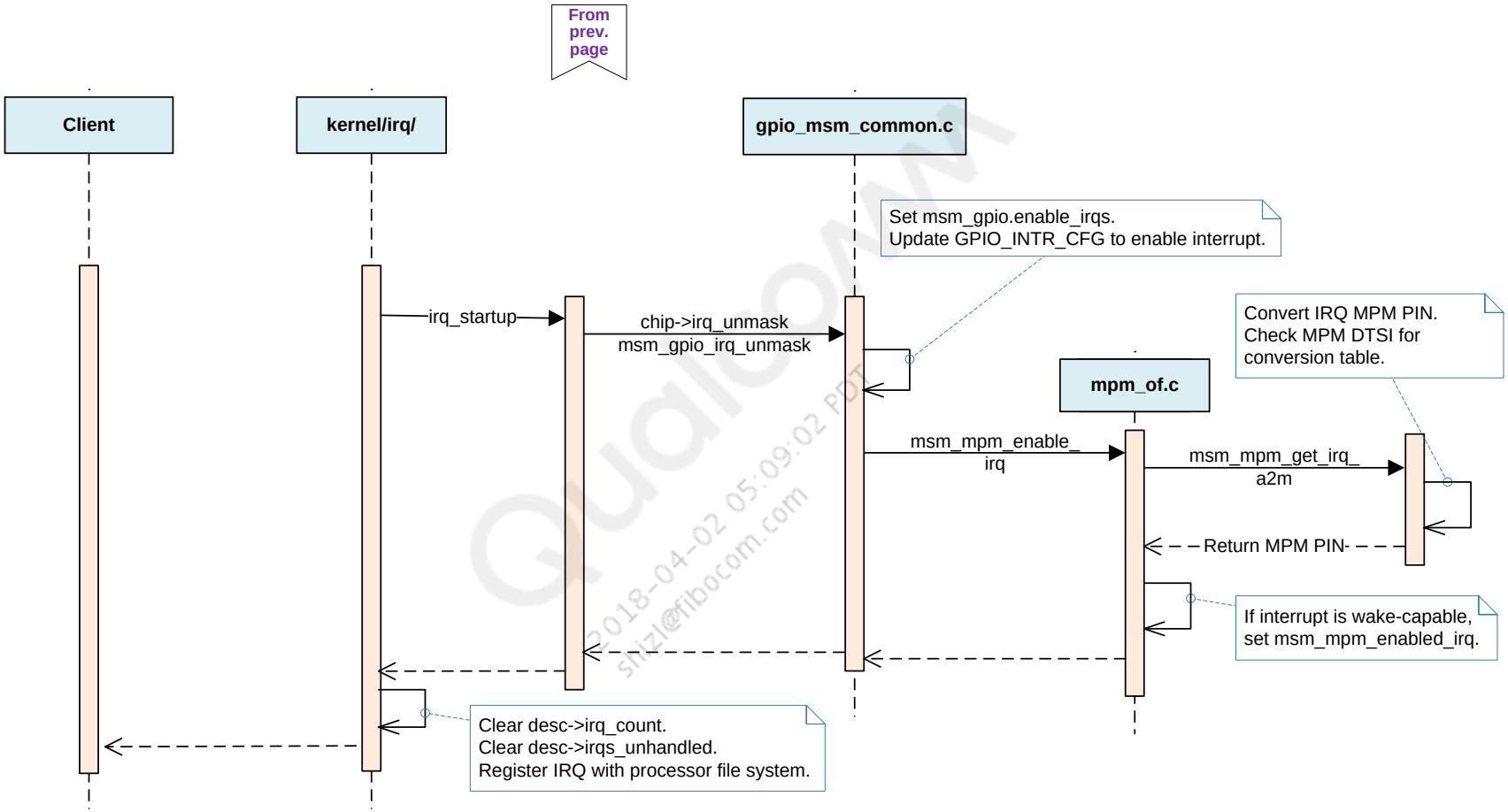


Figure 7-2 Register a GPIO IRQ (2 of 2)

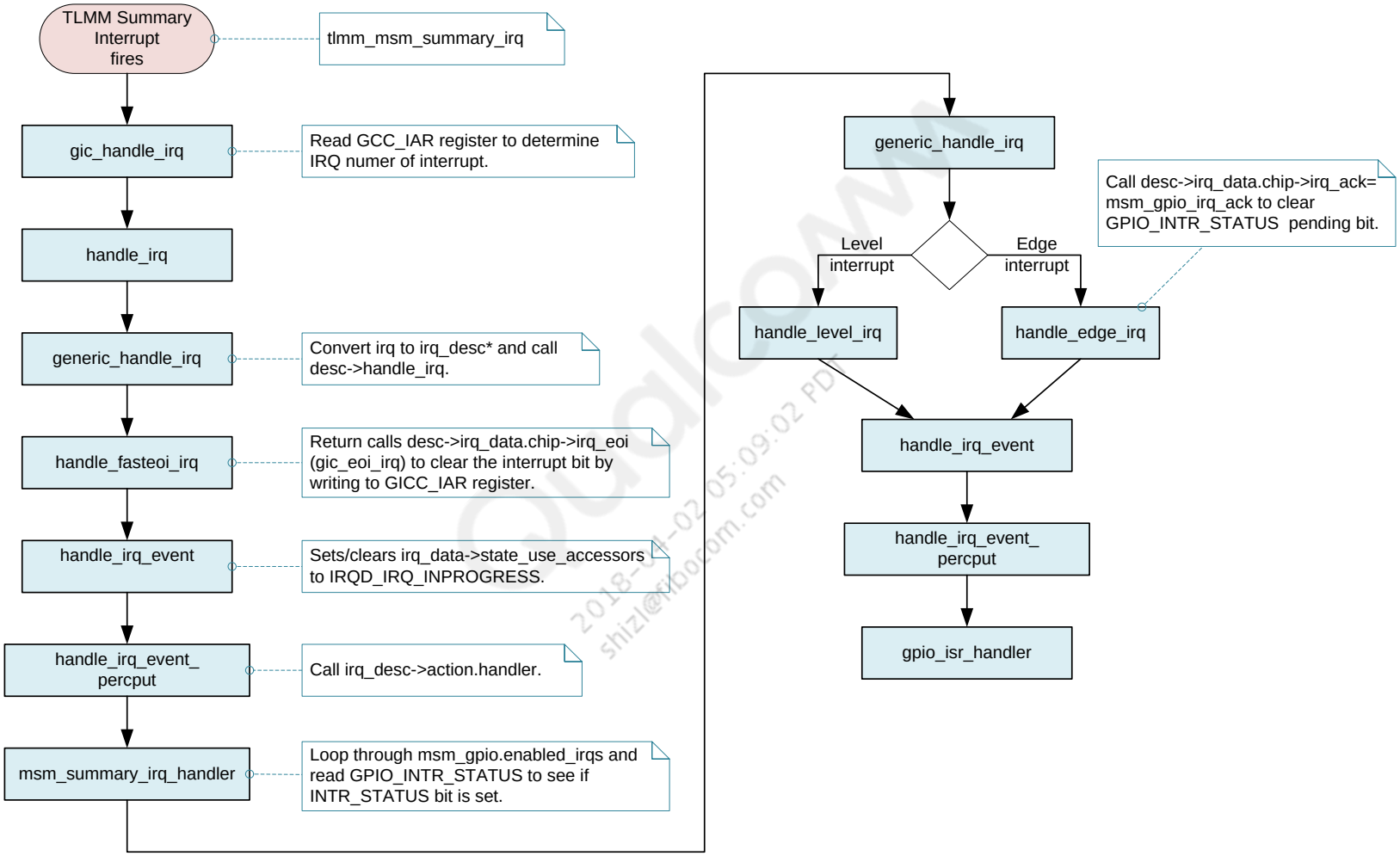


Figure 7-3 Fire a GPIO interrupt

7.3 Check GPIO configuration

Android default command /system/bin/r can be used to check the TLMM_GPIO registers.

The examples use GPIO 12 and are based on the MSM8994 platform.

1. Check GPIO configuration with TLMM_GPIO_CFG_n
   Register – TLMM_GPIO_CFGn, n=[0..145]: 0xFD510000 + 0x00001000 (0xFD511000) + 0x10 * (n)
   For GPIO_12, TLMM_GPIO_CFG12 is: 0xFD510000+0x1000+0x10*0xC = 0xFD5110C0

```
adb  root
adb shell
#/system/bin/r 0xFD5110C0

/system/bin/r 0xFD5110C0
fd5110c0: 00000200

bit[1:0] is0, that means NO-PULL
bit[5:2] is 0, that means FUNC_0 (normal GPIO)
bit[8:6] is 0, that means DRV_2_MA( 2mA drive strength)
bit[9] is 1, that means output enabled(GPIO behaves as output).

 /system/bin/r can be used to change the register's value as well:
     #/sytem/bin/r 0xFD5110C0 0x0     /*Disable the output of GPIO_12,
it makes the GPIO_12 as an input gpio */

 /system/bin/r 0xFD5110C0 0
 fd5110c0: 00000000
```

2. Check GPIO status with TLMM_GPIO_IN_OUT_n
   Register – TLMM_GPIO_IN_OUTn, n=[0..145]: 0xFD510000 + 0x00001004 (0xFD511004) + 0x10 * (n)
   For GPIO_12, TLMM_GPIO_IN_OUT_12 is: 0xFD510000+0x1004+0x10*0xC = 0xFD5110C4

```
# /system/bin/r 0xFD5110C4

/system/bin/r 0xFD5110C4
fd5110c4: 00000000
```

   Bit [1] is 0, which means GPIO_12 is output low.

3. Check GPIO interrupt settings with TLMM_GPIO_INTR_CFG_n
   Register – TLMM_GPIO_IN_OUTn, n=[0..145]: 0xFD510000 + 0x00001004 (0xFD511004) + 0x10 * (n)
   For GPIO_12, TLMM_GPIO_IN_OUT_12 is: 0xFD510000+0x1008+0x10*0xC = 0xFD5110C8

```
# /system/bin/r 0xFD5110C8
```

```
/system/bin/r 0xFD5110C8
fd5110c8: 000000e2
```

Bit[0] is 0, which means interrupt is not enabled for GPIO_12; it is not used as an interrupt pin.

8 BUS scale

The Linux BUS scaling driver provides access for clients to change the performance characteristics of fabric buses. Clients can request certain performance, and the bus scale driver aggregates all of those requirements and programs the hardware accordingly.

The BUS topology differs from chipset to chipset. The client does not need to know the actual topology. The client requests certain performance between the master and slave. The BUS scale driver automatically calculates the topology, and generates the final bus frequencies.

Each bus topology is specified in the DTSI file:
`/kernel/arch/arm/boot/dts/qcom/<chipset>-bus.dtsi}`.

For example:

- `/kernel/arch/arm/boot/dts/qcom/msm8994-bus.dtsi`
- `/kernel/arch/arm/boot/dts/qcom/msm8226-bus.dtsi`

8.1 Sample topology

For BLSP2 to EBI, observe the topology BLSP2 → Peripheral NOC → System NOC → BIMC NOC → EBI.

Figure 8-1 shows the MSM8994 block diagram.

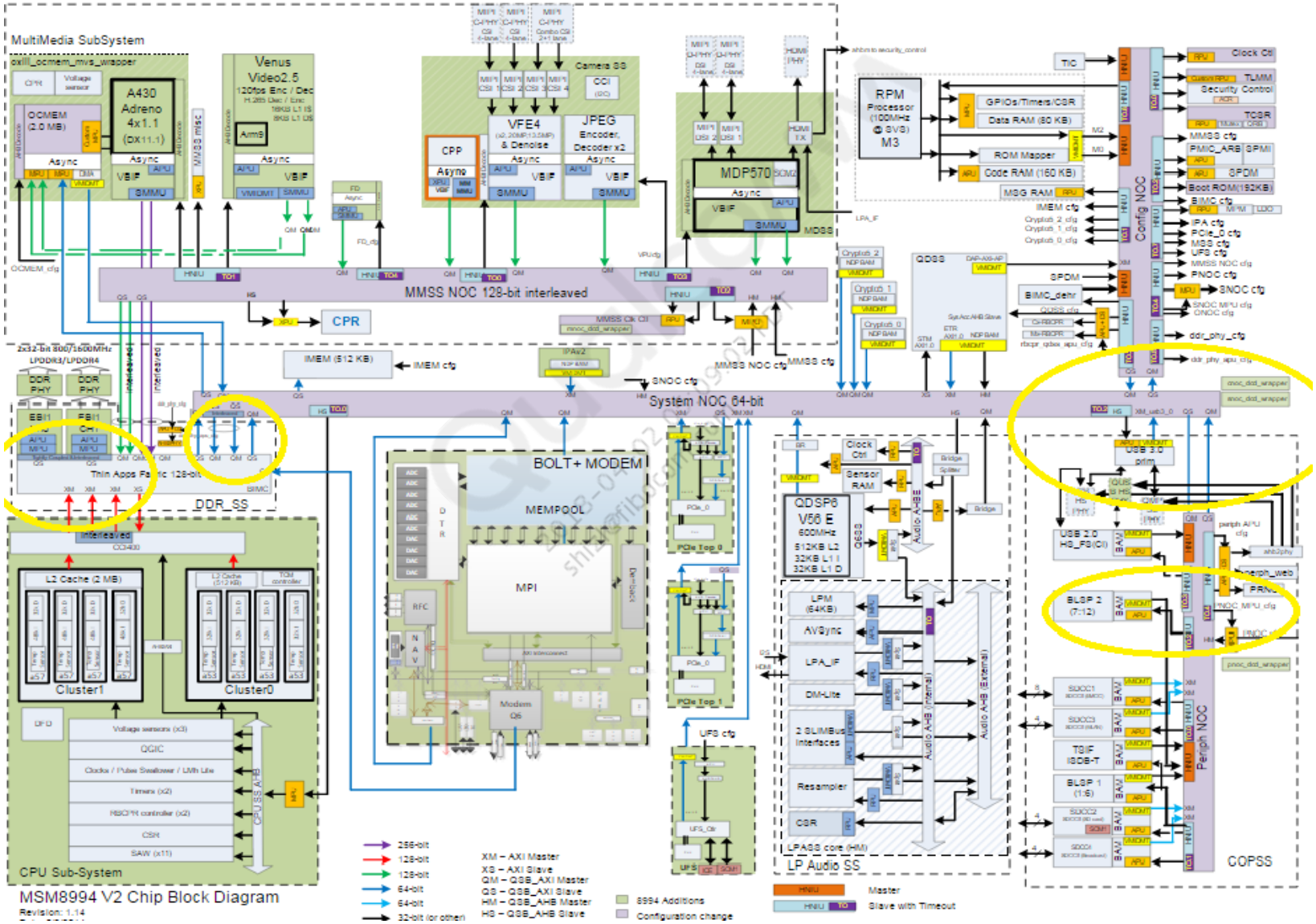


Figure 8-1 MSM8994 block diagram

The highlighted topology is shown in the following source code.

```
\\Above topology showen from msm8974-bus.dtsi
  msm-periph-noc@fc468000 {
    compatible = "msm-bus-fabric";
    reg = <0xfc468000 0x00004000>;
    cell-id = <4096>;
    label = "msm_periph_noc";
    qcom,fabclk-dual = "bus_clk";
    qcom,fabclk-active = "bus_a_clk";
    qcom,ntieredslaves = <0>;
    qcom,hw-sel = "NoC";
    qcom,rpm-en;
    ...
    mas-blsp-2 {
      cell-id = <84>;
      label = "mas-blsp-2";
      qcom,masterp = <6>;
      qcom,tier = <2>;
      qcom,buswidth = <8>;
      qcom,mas-hw-id = <39>;
    };
    ...
    fab-snoc {
      cell-id = <1024>;
      label = "fab-snoc";
      qcom,gateway;
      qcom,slavep = <14>;
      qcom,masterp = <11>;
      qcom,tier = <2>;
      qcom,buswidth = <8>;
      qcom,slv-hw-id = <45>;
      qcom,mas-hw-id = <44>;
    };
    ...
  };
 msm-sys-noc@fc460000 {
    compatible = "msm-bus-fabric";
    reg = <0xfc460000 0x00004000>;
    cell-id = <1024>;
    label = "msm_sys_noc";
    qcom,fabclk-dual = "bus_clk";
    qcom,fabclk-active = "bus_a_clk";
    qcom,ntieredslaves = <0>;
    qcom,qos-freq = <4800>;
    qcom,hw-sel = "NoC";
    qcom,rpm-en;
    ...
    fab-bimc {
      cell-id = <0>;
      label= "fab-bimc";
      qcom,gateway;
      qcom,slavep = <7 8>;
      qcom,masterp = <3>;
      qcom,buswidth = <8>;
      qcom,mas-hw-id = <21>;
```

```
      qcom,slv-hw-id = <24>;
    };
    fab-pnoc {
      cell-id = <4096>;
      label = "fab-pnoc";
      qcom,gateway;
      qcom,slavep = <12>;
      qcom,masterp = <11>;
      qcom,buswidth = <8>;
      qcom,qport = <8>;
      qcom,mas-hw-id = <29>;
      qcom,slv-hw-id = <28>;
      qcom,mode = "Fixed";
      qcom,prio1 = <2>;
      qcom,prio0 = <2>;
    };
    ...
  }
  msm-bimc@0xfc380000 {
    compatible = "msm-bus-fabric";
    reg = <0xfc380000 0x0006A000>;
    cell-id = <0>;
    label = "msm_bimc";
    qcom,fabclk-dual = "mem_clk";
    qcom,fabclk-active = "mem_a_clk";
    qcom,ntieredslaves = <0>;
    qcom,qos-freq = <19200>;
    qcom,hw-sel = "BIMC";
    qcom,rpm-en;
    ...
    fab-snoc {
      cell-id = <1024>;
      label = "fab-snoc";
      qcom,gateway;
      qcom,slavep = <3>;
      qcom,masterp = <5 6>;
      qcom,qport = <5 6>;
      qcom,buswidth = <8>;
      qcom,ws = <10000>;
      qcom,mas-hw-id = <3>;
      qcom,slv-hw-id = <2>;
      qcom,mode = "Bypass";
      qcom,hw-sel = "RPM";
    };
    ...
    slv-ebi-ch0 {
      cell-id = <512>;
      label = "slv-ebi-ch0";
      qcom,slavep = <0 1>;
      qcom,tier = <2>;
      qcom,buswidth = <8>;
      qcom,slv-hw-id = <0>;
      qcom,mode = "Bypass";
    };
  }
```

8.2 Client usage

A client can register with the bus scale driver by adding bus scale nodes into the device tree.

For example:

```
uart7: uart@f995d000 {
  qcom,msm-bus,name = "uart7";
  qcom,msm-bus,num-cases = <2>;
  qcom,msm-bus,num-paths = <1>;
  qcom,msm-bus,vectors-KBps =
  /*
  84 = Master PORT ID (msm8974-bus.dtsi, mas-blsp-2 cell-id)
  512 = Slave Port ID (msm8974-bus.dtsi, slv-ebi-ch0 cell-id)
  Next two digits are Instantaneous bandwidth, and Arbitrated bandwidth in
KBps.
  */
  <84 512 0 0>,
  <84 512 500 800>;
}
```

Functionality available to the client	API function
Parse the device tree	struct msm_bus_scale_pdata *msm_bus_cl_get_pdata(struct platform_device *pdev)
Register the bus scale table	uint32_t msm_bus_scale_register_client(struct msm_bus_scale_pdata *pdata)
Request the bus scale	int msm_bus_scale_client_update_request(uint32_t cl, unsigned int index)
Unregister a bus scale node	msm_bus_scale_unregister_client()

The following example illustrates requesting the bus scale:

```
msm_bus_scale_client_update_request()-->
  update_path()-->Calculate the clock rates
    msm_bus_fabric_update_clks()-->Sets the clock rates
  <---
  bus_for_each_dev()-->
    msm_bus_commit_fn()-->
      msm_bus_fabric_hw_commit()
```

8.3 Verification

If the client is registered, it can verify the bus scale by checking the debugfs file system.

1. List all of the registered clients.

```
shell@msm8974:/sys/kernel/debug/msm-bus-dbg/client-data # ls
ls
acpuclk-8974
audio-ocmem
f9923000.spi
f9924000.i2c
f9967000.i2c
grp3d
mdss_mdp
mdss_pp
msm-rng-noc
ocimem_snoc
pil-venus
qcedev-noc
qcom,dec-ddr-ab-ib
qcom,dec-ocmem-ab-ib
qcom,enc-ddr-ab-ib
qcom,enc-ocmem-ab-ib
qseecom-noc
scm_pas
sdhc1
sdhc2
serial_uart2
uart7
update-request
usb3
```

2. Monitor the client bus scale activity by running the following command:

```
cat /sys/kernel/debug/msm-bus-dbg/client-data/client_name
```

For example:

```
shell@msm8974:/ # cat /sys/kernel/debug/msm-bus-dbg/client-data/uart7
cat /sys/kernel/debug/msm-bus-dbg/client-data/uart7

0.955193492 //Client requests
curr   : 1
masters: 84
slaves : 512
ab     : 500000
ib     : 800000

0.958651356
curr   : 0
masters: 84
slaves : 512
ab     : 0
ib     : 0

49.821066393
curr   : 1
masters: 84
slaves : 512
ab     : 500000
ib     : 800000

52.594351964
curr   : 0
masters: 84
slaves : 512
ab     : 0
ib     : 0

54.592914984
curr   : 1
masters: 84
slaves : 512
ab     : 500000
ib     : 800000

57.416415816
curr   : 0
```

```
masters: 84
slaves : 512
ab     : 0
ib     : 0
shell@msm8974:/ #
```

3. Enable debug at runtime or compile time to determine if the clock topology is correct. To enable debug compile time:

```
diff --git a/arch/arm/mach-msm/msm_bus/msm_bus_core.h
           b/arch/arm/mach-msm/msm_bus/msm_bus_core.h
index 7e4a513..e4be763 100644
--- a/arch/arm/mach-msm/msm_bus/msm_bus_core.h
+++ b/arch/arm/mach-msm/msm_bus/msm_bus_core.h
@@ -21,7 +21,7 @@
 #include <mach/msm_bus.h>

 #define MSM_BUS_DBG(msg, ...) \
-       pr_debug(msg, ## __VA_ARGS__)
+       pr_info(msg, ## __VA_ARGS__)
 #define MSM_BUS_ERR(msg, ...) \
        pr_err(msg, ## __VA_ARGS__)
 #define MSM_BUS_WARN(msg, ...) \
```

Example of a log request between Master-84 (BLSP-2) and Slave (512)

```
<6>[  830.394205] AXI: msm_bus_scale_client_update_request(): cl:
4119547328 index: 1 curr: 0 num_paths: 1
<6>[  830.394291] AXI: msm_bus_scale_client_update_request(): ab: 0 ib: 0
<6>[  830.394348] AXI: update_path(): args: 4180 4180 0 800000 500000 0 0 0
<6>[  830.394382] AXI: update_path(): Client passed index :0
<6>[  830.394428] AXI: update_path(): id: 4180
<6>[  830.394458] AXI: update_path(): id:4180, next: 1024
//Updating PNOC - SNOC
<6>[  830.394496] AXI: update_path(): fabric: 4096 slave: 1024, slave-
width: 8 info: 4180
<6>[  830.394534] AXI: NOC: msm_bus_noc_update_bw(): NOC: No qos frequency
to update bw
<6>[  830.394574] AXI: update_path(): AXI: Hop: 1024, ports: 1, bwsum_hz:
125000
<6>[  830.394609] AXI: update_path(): up-clk: curr_hz: 0, req_hz: 800000,
bw_hz 125000
<6>[  830.394648] AXI: msm_bus_fabric_update_clks(): max_pclk from
gateways: 125000
<6>[  830.394705] AXI: msm_bus_fabric_update_clks(): max_pclk from slaves
   gws: 125000
```

```
<6>[  830.394743] AXI: msm_bus_fabric_update_clks(): clks: id: 4096 set-
clk: 125000 bws_hz:125000
<6>[  830.394785] AXI: update_path(): id: 1024
//Updating SNOC - BIMC
<6>[  830.394814] AXI: update_path(): id:1024, next: 0
<6>[  830.394851] AXI: update_path(): fabric: 1024 slave: 0, slave-width: 8
info: 1024
<6>[  830.394889] AXI: NOC: msm_bus_noc_update_bw(): NOC: Update bw for:
1024: 500000
<6>[  830.394929] AXI: NOC: msm_bus_noc_update_bw(): NOC: Update mas_bw:
ID: 1024, BW: 1000000 ports:1
<6>[  830.394969] AXI: NOC: msm_bus_noc_update_bw(): NOC: Update slave_bw
for ID: 0 -> 120500000
<6>[  830.395006] AXI: NOC: msm_bus_noc_update_bw(): NOC: Update slave_bw
for hw_id: 24, index: 7
<6>[  830.395044] AXI: NOC: msm_bus_noc_update_bw(): NOC: Update slave_bw
for ID: 0 -> 120500000
<6>[  830.395080] AXI: NOC: msm_bus_noc_update_bw(): NOC: Update slave_bw
for hw_id: 24, index: 8
<6>[  830.395119] AXI: update_path(): AXI: Hop: 0, ports: 2, bwsum_hz:
15062500
<6>[  830.395154] AXI: update_path(): up-clk: curr_hz: 0, req_hz: 800000,
bw_hz 15062500
<6>[  830.395194] AXI: msm_bus_fabric_update_clks(): max_pclk from
gateways: 120000000
<6>[  830.395245] AXI: msm_bus_fabric_update_clks(): max_pclk from slaves
   gws: 120000000
<6>[  830.395283] AXI: msm_bus_fabric_update_clks(): clks: id: 1024 set-
clk: 120000000 bws_hz:15062500
<6>[  830.395333] AXI: update_path(): id: 0
//Updating BIMC - DDR
<6>[  830.395361] AXI: update_path(): id:0, next: 512
<6>[  830.395398] AXI: update_path(): fabric: 0 slave: 512, slave-width: 8
info: 0
<6>[  830.395437] AXI: BIMC: msm_bus_bimc_update_bw(): BIMC: Update bw for
ID 0, with IID: 0: 500000
<6>[  830.395477] AXI: BIMC: msm_bus_bimc_update_bw(): BIMC: Update mas_bw
for ID: 0 -> 241000000
<6>[  830.395514] AXI: BIMC: msm_bus_bimc_update_bw(): BIMC: ID: 512,
Sports: 2
<6>[  830.395551] AXI: BIMC: msm_bus_bimc_update_bw(): BIMC: Update
slave_bw: ID: 512 -> 120500000
<6>[  830.395586] AXI: BIMC: msm_bus_bimc_update_bw(): BIMC: Update
slave_bw: index: 0
<6>[  830.395621] AXI: BIMC: msm_bus_bimc_update_bw(): Slave dirty: 512,
slavep: 0
<6>[  830.395655] AXI: BIMC: msm_bus_bimc_update_bw(): BIMC: Update
slave_bw: ID: 512 -> 120500000
```

```
<6>[  830.395690] AXI: BIMC: msm_bus_bimc_update_bw(): BIMC: Update
slave_bw: index: 1
<6>[  830.395725] AXI: BIMC: msm_bus_bimc_update_bw(): Slave dirty: 512,
slavep: 1
```

9 FAQs

9.1 General information

QUESTION 1 How many UART/I2C/SPI are supported in my platform, and which GPIOs are corresponded?

ANSWER Refer to the *BLSP configuration table* in the hardware device specification document of your specified platform.

QUESTION 2 Can I use the same BLSP QUP core as both I2C and SPI?

ANSWER No. Because SPI and I2C shares the same QUP core, you can use it as SPI or I2C.

QUESTION 3 What is the constraint when I use BLSP UART/I2C/SPI High Speed mode?

ANSWER In High Speed mode, the peripheral makes use of BAM for data transfer. Hence, confirm the ownership of the correspond BAM PIPEs assigned to the subsystem that uses the BLSP UART/I2C/SPI from TrustZone. Failure to do so, results in XPU protection error.

9.2 UART

QUESTION 1 How can I configure a BLSP UART in LK and kernel?

ANSWER Refer to Chapter 3.

QUESTION 2 Can we use UART to output log in SBL stage?

ANSWER Yes. SBL has the debug UART port, but for specified platforms and also how to enable it. For assistance, submit a case to Qualcomm Technologies, Inc. (QTI) at https://createpoint.qti.qualcomm.com/ and select bootloader as the problem area.

QUESTION 3 What's the difference between driver code msm_serial_hs.c and msm_serial_hs_lite.c?

ANSWER Refer to Section 3.3.

QUESTION 4 Which baud rates are supported in my UART driver?

ANSWER Refer to Section 3.1.

QUESTION 5 How to change baud rate in my application?

ANSWER Refer to any Linux cook book related with terminal IO for the standard APIs. For example, `cfsetospeed, and cfsetispeed.`

```
options.c_cflag &= ~CSIZE;
options.c_cflag |= (CS8 | CLOCAL | CREAD | CRTSCTS);
options.c_iflag = IGNPAR;
options.c_oflag = 0;
options.c_lflag = 0;
cfsetospeed(&options, speed);
cfsetispeed(&options, speed);
```

QUESTION 6 Why does not my device enter Low Power mode after HS UART is enabled?

ANSWER Refer to Section 3.5.3. For HS UART, OEM's client driver needs to maintain the clocks by using the APIs provided by our driver. If OEM does not turn off the clocks in idle state, the on clock will vote against system pm.

9.3 I2C

QUESTION 1 How to configure a BLSP I2C in LK and kernel?

ANSWER Refer to Chapter 4.

QUESTION 2 How to enable DMA transfer of I2C in kernel?

ANSWER DMA mode will be enabled automatically when data pay load is larger than 48 bytes.

QUESTION 3 What's the max transfer rate and features supported by our BLSP I2C?

ANSWER Refer to Section 4.1.

QUESTION 4 The I2C waveform does not follow the I2C specification, especially i2c_low_time, and i2c_high time. How can I tune it?

ANSWER Refer to Section 4.3.6 of this document to adjust clock dividers.

QUESTION 5 How to debug my I2C issues?

ANSWER Refer to Section 4.6.

9.4 SPI

NOTE: Numerous changes were made in this section.

QUESTION 1 How to configure a BLSP SPI kernel?

ANSWER Refer to Chapter 5 of this document.

QUESTION 2 How to configure BLSP SPI in LK?

ANSWER BLSP SPI is not supported in LK and in short term, we don't have plan to support it in LK.

QUESTION 3 Which transfer rates are supported by SPI driver?

ANSWER Try to check the clock table defined in the source code mentioned in Section 5.2.1.

QUESTION 4 How to enable BAM transfer mode?

ANSWER BAM transfer will be enabled automatically after the following conditions are met.

Defines BAM related resources in SPI driver device tree following Section 5.2.1

SPI message meets following requirement:

Transfer size is greater than 3*block size that's 48 bytes.

Data payload buffers are aligned to cache line.

Bytes-per-word is 8, 16 or 32.

QUESTION 5 Whether multi-CS is supported for all BLSP SPI?

ANSWER No, only specified BLSP SPI support multi-CS, refer to the hardware specification document of your specified platform.

QUESTION 6 How to enable multi-CS?

ANSWER Refer to Section 5.2.3.

QUESTION 7 How to measure SPI throughput?

ANSWER Refer to Section 5.4

QUESTION 8 Why N=32, big payload size are suggested for throughput measurement?

ANSWER N=32 can save (2~3)*3 clock cycles idle time for each 4 bytes, and big payload size can weak the constant hardware setup time side effect

QUESTION 9 How can we measure SPI throughput if ftrace is not enabled in my platform?

ANSWER Try to make use of OS time APIs:ktime_get to get 2 timestamps before and after SPI transfer, and measure the time delta.

A References

A.1 Related documents

Documents	
Qualcomm Technologies, Inc.	
Low-Speed Peripherals Overview	80-NA157-24

A.2 Acronyms and terms

Acronyms or Term	Definition
ADM	Application Data Mover
AHB	AMBA Advanced High-Performance Bus
BAM	Bus Access Manager
BLSP	BAM Low-Speed Peripheral
CDP	Core Development Platform
CS	Chip Select
CTS	Clear-to-Send
DMA	Direct Memory Access
DTB	Device Tree Blob
DTC	DTS Compiler Tool
DTS	Device Tree Source
EOT	End-of-Transfer
GSBI	General Serial Bus Interface
I2C	Inter-Integrated Circuit
IrDA	Infrared Data Association
LK	Little Kernel
MTP	Modem Test Platform
PNoC	Peripheral Network on a Chip
QUP	Qualcomm Universal Peripheral (Serial)
RFR	Ready for Receiving
SDCC	Secure Digital Card Controller
SPI	Serial Peripheral Interface
SPS	Smart Peripheral Subsystem
UART	Universal Asynchronous Receiver/Transmitter
UIM	User Identity Module